

SSA Translation is an Abstract Interpretation

MATTHIEU LEMERRE, Université Paris-Saclay, CEA, List, France

Static single assignment (SSA) form is a popular intermediate representation that helps implement useful static analyses, including global value numbering (GVN), sparse dataflow analyses, or SMT-based abstract interpretation or model checking. However, the precision of the SSA translation itself depends on static analyses, and a priori static analysis is even indispensable in the case of low-level input languages like machine code. To solve this chicken-and-egg problem, we propose to turn the SSA translation into a standard static analysis based on abstract interpretation. This allows the SSA translation to be combined with other static analyses in a single pass, taking advantage of the fact that it is more precise to combine analyses than applying passes in sequence. We illustrate the practicality of these results by writing a simple dataflow analysis that performs SSA translation, optimistic global value numbering, sparse conditional constant propagation, and loop-invariant code motion in a single small pass; and by presenting a multi-language static analyzer for both C and machine code that uses the SSA abstract domain as its main intermediate representation.

CCS Concepts: • **Software and its engineering** → **Compilers**; *Formal software verification*; • **Theory of computation** → **Program analysis**; **Program verification**; **Abstraction**; *Equational logic and rewriting*.

Additional Key Words and Phrases: Static Single Assignment (SSA), Abstract interpretation, Cyclic term graph.

ACM Reference Format:

Matthieu Lemerre. 2023. SSA Translation is an Abstract Interpretation. *Proc. ACM Program. Lang.* 7, POPL, Article 65 (January 2023), 41 pages. <https://doi.org/10.1145/3571258>

1 INTRODUCTION

"SSA translation is an abstract interpretation" may be a surprising statement. Usually, SSA translation [Cytron et al. 1991] is presented as a code transformation that modifies the input program so that each variable is assigned only once (the Static Single Assignment (SSA) property). Abstract interpretation, on the other hand, is a method for designing sound static analyses that retrieve information about the program. If a code transformation can be based on the results of an analysis, and the results of an analysis can be made more precise by a previous code transformation, both are usually clearly separated and done in separate passes. This is unfortunate as it is known that performing analyses in a sequence of passes yields less precise results than combining them in a single pass [Click and Cooper 1995; Cousot and Cousot 1979].

To solve this problem, we perform SSA translation using abstract interpretation, i.e. the program abstraction that the analysis computes is the SSA translation of the program. More precisely, our SSA abstraction builds upon a variant of a global value numbering (GVN) analysis that we call *symbolic expression* analysis (Section 4). This is unlike most existing work (e.g., [Alpern et al. 1988; Barthe et al. 2014; Rosen et al. 1988; Rüthing et al. 1999]), where GVN is performed *after* SSA translation. Indeed in existing work, GVN requires a previous SSA translation to name the subexpressions of the program so that GVN then only needs to partition the names in equality classes. By contrast, in our work, the underlying abstraction used by GVN is not SSA names, but a

Author's address: Matthieu Lemerre, Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France, matthieu.lemerre@cea.fr.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2023 Copyright held by the owner/author(s).

2475-1421/2023/1-ART65

<https://doi.org/10.1145/3571258>

50 cyclic term graph [Ariola and Klop 1996] called the global value graph [Reif and Lewis 1986]. This
51 allows performing GVN directly on the original program.

52 Our SSA graph abstraction (Section 5) then builds upon the global value graph abstraction by
53 adding control flow and conditionals. Like the symbolic expression analysis, the SSA translation
54 is described as a standard dataflow analysis that directly applies to the source program. As SSA
55 translation preserves the semantics of the original program, we prove that our SSA-abstraction
56 analysis is both sound and complete. Interestingly, our proof method consists in finding a surjective
57 strong homomorphism (a stronger variant of bisimulation) between the SSA translation and the
58 original program, where the homomorphism directly derives from our symbolic expression analysis.
59 Finally, we provide (Section 6) alternative transfer functions for our SSA translation that allows the
60 analysis to be complete (i.e. excludes spurious behaviors) also on unfinished fixpoint computations,
61 which makes our SSA translation algorithm incremental.

62 The syntax and semantics of our SSA abstraction are non-standard but can readily be translated
63 to the usual SSA syntax by choosing an order of evaluation for expressions; moreover, our SSA
64 translation enjoys the classic minimality and strictness [Rastello 2016] properties of SSA.

65
66 *Motivation.* Interesting theoretical contributions of this work are its alternative syntax and
67 semantics for SSA form (both are new, concise and simple), the algorithm used for its translation
68 (a standard dataflow analysis which builds on global value numbering and does not require the
69 computation of any dominance properties), the theoretical connections it reveals between global
70 value numbering, symbolic expression analysis and SSA, and the technique used to perform dataflow
71 analysis on cyclic terms or systems of recursive equations in general (reusing cyclic names of the
72 original program to detect cycles and guarantee termination).

73 A practical motivation for presenting SSA translation as an abstract domain is that abstract
74 domains can be combined modularly [Cousot and Cousot 1979; Cousot et al. 2006; Journault et al.
75 2019]. An application of this is to explore alternative designs for the combination of compiler
76 optimizations. In general, compiler optimizations are written as a succession of analysis and
77 rewriting passes. Viewing SSA as an abstract domain allows combining the analysis and the
78 rewriting pass in a single step, with the benefit that the SSA invariants (e.g., related to domination)
79 are automatically maintained when the control-flow graph changes (e.g., when a branch is found
80 dead, or a new target of a computed goto is discovered). This is beneficial as “maintaining SSA is
81 often one of the more complicated and error-prone parts in such optimizations” [Hack 2016].

82 It is even possible to group all the different analyses and transformations in a single pass,
83 avoiding the burden of performing any intermediate rewriting steps. As an example (Section 7),
84 we implemented a simple dataflow analysis that performs global value numbering, loop-invariant
85 code motion, sparse conditional constant propagation, and SSA translation in a single dataflow
86 analysis pass. This pass is followed by a pass of linear complexity translating our internal SSA
87 representation into LLVM bytecode, mainly consisting in making explicit the order of evaluation of
88 the symbolic expressions appearing in our representation of SSA. Experimental results show that
89 the implementation is small and the compiling time is compatible with inclusion in a compiler.

90 Another application is formal verification. In general, combining analyses yields an analysis
91 which is more precise than performing different analyses in a sequence of passes [Click and Cooper
92 1995; Cousot and Cousot 1979]. Viewing SSA as an abstract domain is thus interesting when
93 analyzing programs where the SSA translation can be improved by a static analysis, and the static
94 analysis can itself be improved by more precise SSA translation. For instance, it is common to use
95 SSA as an intermediate language before translation into logic formulas queried to an automated
96 solver (e.g., [Brain et al. 2015; Gurfinkel et al. 2015; Henry et al. 2012; Leino 2005]). Combining
97 the SSA translation with other abstract domains allows simplifying the generated formula, and
98

simplifications significantly impact the solver performance [Farinier et al. 2018; Gurfinkel et al. 2015]. In turn, the solver's response can improve the precision of the static analysis. One example application is the analysis of machine code, where the precision of the recovered control-flow graph is crucial, and SMT-based techniques can be used to ensure the reachability of the locations of the recovered CFG [Djoudi et al. 2016; Reinbacher and Brauer 2011]. Note, however, that using such a combination requires that the SSA form produced from an unfinished fixpoint iteration is complete (i.e., does not introduce any spurious behavior): this problem is solved in Section 6.

The benefits of this combination of domains grow in importance when analyzing lower-level languages, such as machine code or C. Indeed, more precise SSA translation in these languages demands that memory locations are promoted to SSA variables; this promotion requires precise memory and control flow analyses, which depend on a numeric analysis; the numeric analysis can be enhanced by more precise SSA translation. Having SSA as an abstract domain and using it with a combination of memory, numerical and control-flow abstractions solves this chicken-and-egg problem. For instance, we have implemented a static analyzer for both C and machine code programs that decompiles low-level code into a higher-level SSA abstraction using the SSA graph abstract domain; simultaneously, this SSA abstraction is used as the language on which the numerical analyses are performed. This paper is the first step in explaining how such an analysis works.

2 OVERVIEW

This section provides an overview of our main results on a running example (Figure 1b) that we will use in most of the paper. In our view, the essence of SSA is the addition of control flow information to a structure called the global value graph [Reif and Lewis 1986]; our presentation shows how to build these two abstractions in turn. We start from a program consisting of basic blocks, each associated with a unique program *location* $\ell_0, \dots, \ell_3 \in \mathbb{L}$ labeling the basic block. Each block contains a sequence of instructions and performs nondeterministic jumps to other basic blocks; these jumps have guards that block some executions. The semantic is standard and consists of a transition between states, where a state contains both the current location and a *store* $\sigma \in \Sigma$ (a mapping from *program variables* $x \in \mathbb{X}$ to values). The initial states are all the states starting at ℓ_0 .

2.1 Symbolic expressions, global value numbering and the global value graph.

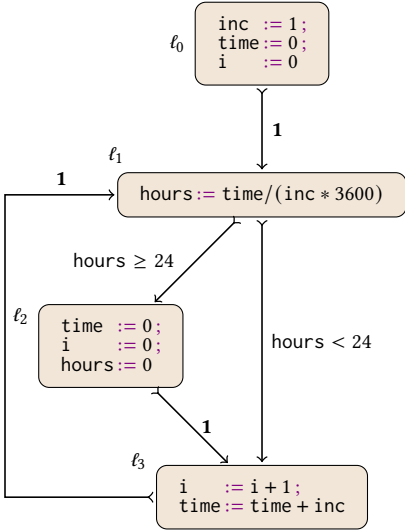
The symbolic expression abstraction. Our main program abstraction $p^\# \in \mathbb{L} \rightarrow \Sigma_\perp^\#$ associates to each location $\ell \in \mathbb{L}$ an *abstract store* $\sigma^\# \in \Sigma_\perp^\#$ describing the possible states before execution of the basic block. An abstract store is either \perp or a mapping from program variables to *symbolic expressions*, i.e. terms containing *symbolic variables* $\in \mathbb{V}$. The result of the analysis for our example program is given on the first line of Figure 1a. For instance, the abstract store at $p^\#[\ell_2]$ refers to two symbolic variables, namely i_{ℓ_1} and time_{ℓ_1} . An abstract store represents all the concrete stores that can be obtained using any valuation of the symbolic variables, e.g., the concrete store $[\text{inc} \mapsto 1, \text{time} \mapsto 0, i \mapsto 0, \text{hours} \mapsto 0]$ is represented by $p^\#[\ell_2]$ (using any valuation where both i_{ℓ_1} and time_{ℓ_1} are assigned 0). Note that the symbolic expressions allow describing relations between variables (e.g. $\text{hours} = \text{time} / (1 * 3600)$ in the states at ℓ_2) or some numerical information (e.g. $2 * x_{\ell_5}$ describes an even value). One particular relation between variables is equality, and indeed this abstraction can be used to detect equality between variables corresponding to *global value numbering* [Alpern et al. 1988; Rosen et al. 1988] (where the "number" uniquely identifies a subterm of a symbolic expression).

Note that symbolic variables $\in \mathbb{V} = \mathbb{X} \times \mathbb{L}$ are named using a program variable and a location: intuitively, i_{ℓ_1} represents the value of program variable i at location ℓ_1 (in the SSA abstraction, as i_{ℓ_1} is unconstrained in the symbolic expression abstraction). To express more equalities, we can reuse

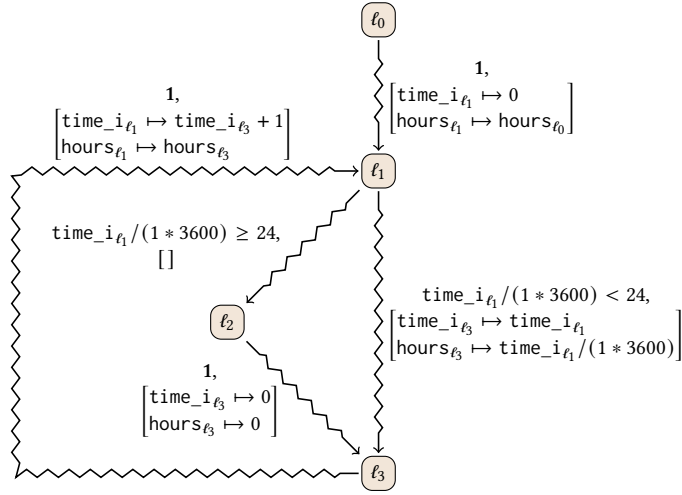
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196

$$\begin{aligned}
 p^\# \text{ when } \mathbb{V} = \mathbb{X} \times \mathbb{L} & \quad \begin{array}{c} \ell_0 \\ \left[\begin{array}{l} \text{inc} \mapsto \text{inc}_{\ell_0} \\ \text{time} \mapsto \text{time}_{\ell_0} \\ i \mapsto i_{\ell_0} \\ \text{hours} \mapsto \text{hours}_{\ell_0} \end{array} \right] \end{array} \quad \begin{array}{c} \ell_1 \\ \left[\begin{array}{l} \text{inc} \mapsto 1 \\ \text{time} \mapsto \text{time}_{\ell_1} \\ i \mapsto i_{\ell_1} \\ \text{hours} \mapsto \text{hours}_{\ell_1} \end{array} \right] \end{array} \quad \begin{array}{c} \ell_2 \\ \left[\begin{array}{l} \text{inc} \mapsto 1 \\ \text{time} \mapsto \text{time}_{\ell_2} \\ i \mapsto i_{\ell_2} \\ \text{hours} \mapsto \text{time}_{\ell_2} / (1 * 3600) \end{array} \right] \end{array} \quad \begin{array}{c} \ell_3 \\ \left[\begin{array}{l} \text{inc} \mapsto 1 \\ \text{time} \mapsto \text{time}_{\ell_3} \\ i \mapsto i_{\ell_3} \\ \text{hours} \mapsto \text{hours}_{\ell_3} \end{array} \right] \end{array} \\
 p^\# \text{ when } \mathbb{V} = \mathcal{P}(\mathbb{X}) \times \mathbb{L} & \quad \begin{array}{c} \left[\begin{array}{l} \text{inc} \mapsto \text{inc}_{\ell_0} \\ \text{time} \mapsto \text{time}_{\ell_0} \\ i \mapsto i_{\ell_0} \\ \text{hours} \mapsto \text{hours}_{\ell_0} \end{array} \right] \end{array} \quad \begin{array}{c} \left[\begin{array}{l} \text{inc} \mapsto 1 \\ \text{time} \mapsto \text{time}_{\ell_1} \\ i \mapsto \text{time}_{\ell_1} \\ \text{hours} \mapsto \text{hours}_{\ell_1} \end{array} \right] \end{array} \quad \begin{array}{c} \left[\begin{array}{l} \text{inc} \mapsto 1 \\ \text{time} \mapsto \text{time}_{\ell_2} \\ i \mapsto \text{time}_{\ell_2} \\ \text{hours} \mapsto \text{time}_{\ell_2} / (1 * 3600) \end{array} \right] \end{array} \quad \begin{array}{c} \left[\begin{array}{l} \text{inc} \mapsto 1 \\ \text{time} \mapsto \text{time}_{\ell_3} \\ i \mapsto \text{time}_{\ell_3} \\ \text{hours} \mapsto \text{hours}_{\ell_3} \end{array} \right] \end{array}
 \end{aligned}$$

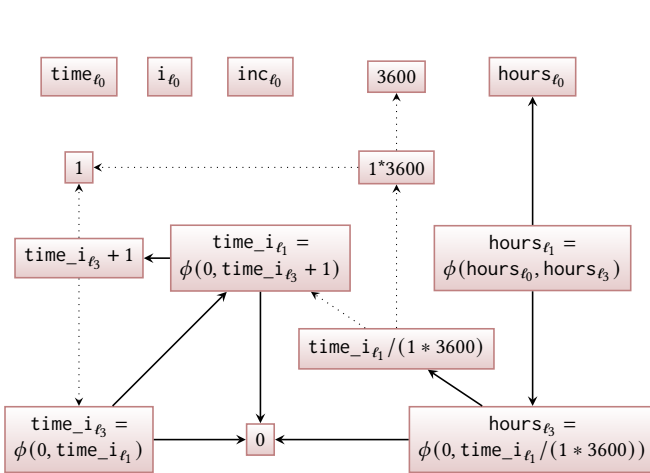
(a) Results of the basic (top) and improved (bottom) symbolic expression analyses (abstract domain: $\mathbb{L} \rightarrow \Sigma_{\perp}^{\#}$).



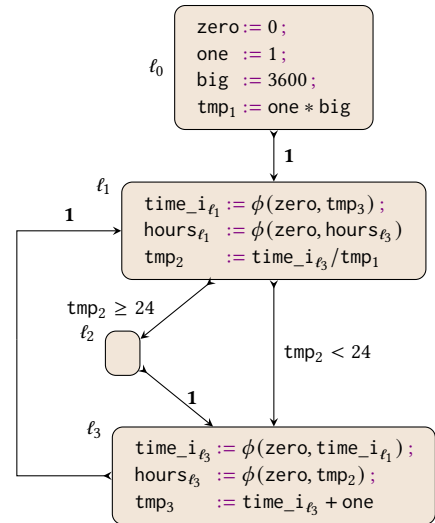
(b) Original program.



(c) Representation in the SSA abstract domain.



(d) The global value graph is a cyclic term graph.



(e) Usual SSA representation.

Fig. 1. Analyses and transformations of an SSA program.

197 the same name when symbolic variables are equal. This is why we shift to another representation
 198 $\mathbb{V} = \mathcal{P}(\mathbb{X}) \times \mathbb{L}$ where a symbolic variable is named using one location and a *set* of program variables;
 199 intuitively, time_i_{ℓ_1} represents the value of both i and time at location ℓ_1 (thus, $i = \text{time}$ at ℓ_1).
 200 The second line of Figure 1a presents the program abstraction $p^\#$ with this change.

201
 202 *Analysis operations.* We focus now our attention on how to compute $p^\#$. Many domain operations
 203 are simple and exact: notably, interpretation of expression substitutes program variables with
 204 their corresponding symbolic expression; interpretation of assignment updates the abstract store
 205 (observe, for instance, how hours is bound to $\text{time_i}_{\ell_1} / (1 * 3600)$ in $p^\#[\ell_2]$ given $p^\#[\ell_1]$). The only
 206 operation in this domain that lose precision is the join operation \sqcup^ℓ , which merges abstract stores
 207 at control-flow join points such as ℓ_3 .

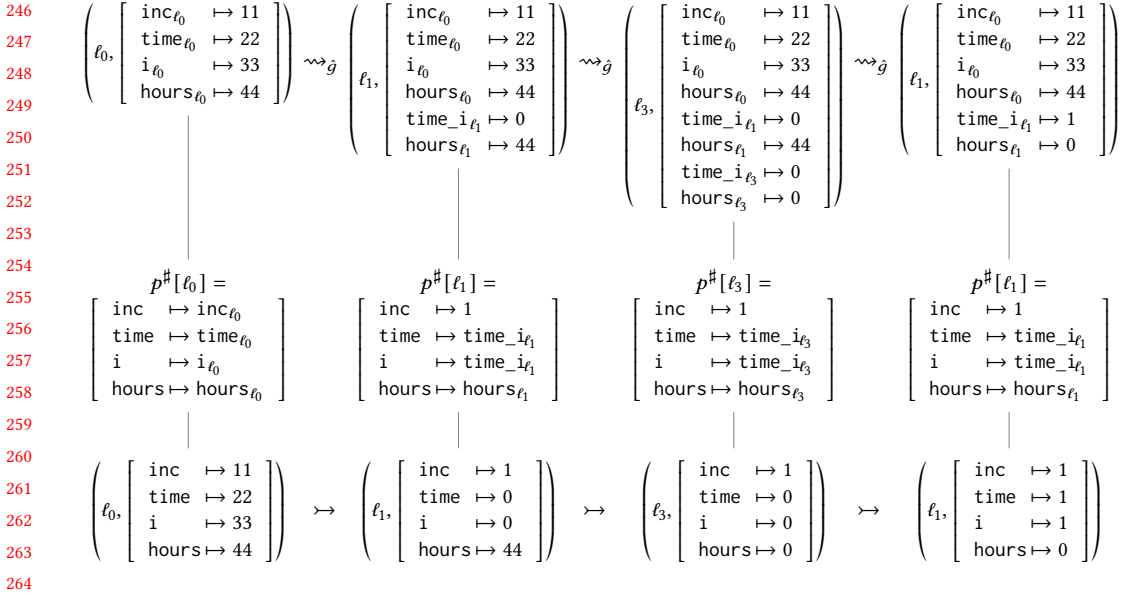
208 One way to perform these joins would be to add a *nondeterministic choice operator* ϕ , as a function
 209 symbol in the language of symbolic expressions. For instance, the value of the program variable i at
 210 ℓ_3 , which is either 0 or time_i_{ℓ_1} depending on the previous program location, could be represented
 211 by the term $\phi(0, \text{time_i}_{\ell_1})$.

212 A first issue with this technique is the handling of nondeterminism: i.e. two terms $\phi(0, 1)$ and
 213 $\phi(0, 1)$ correspond to the same value if one is copied from the other but to different values if
 214 different nondeterministic choices between 0 and 1 must be made in the program. However, this
 215 problem could be mitigated by using different labels for each ϕ function (as in [Alpern et al. 1988;
 216 Rütting et al. 1999]). A much more problematic issue is that this approach cannot terminate in the
 217 presence of loops, as each iteration would create a larger term without ever converging.

218
 219 *The global value graph.* Our solution to both of these issues is to use cyclic term graphs [Ariola
 220 and Klop 1996] instead of simple terms. In cyclic term graphs, recursion variables are introduced to
 221 name some nodes, which allows breaking cycles (in addition, sharing between terms is maximized).
 222 The whole cyclic graph is described using a set of recursive definitions, each definition mapping a
 223 recursion variable to a simple term that can refer to recursion variables. We apply this idea to our
 224 terms: the cyclic term graph corresponds to a structure called the *global value graph* [Reif and Lewis
 225 1986] (sometimes also called the SSA graph [Stanier 2016]), and our symbolic variables play the
 226 role of the recursion variables of the cyclic term graph. In addition, symbolic variables are defined
 227 for all ϕ terms: i.e. every term $\phi(e_1, e_2)$ is named using a symbolic variable var_ℓ . The global value
 228 graph corresponding to the program of Figure 1b is represented graphically in Figure 1d. In this
 229 graph, each node corresponds to a term; dotted lines correspond to the "direct subterm" relation
 230 (e.g., " $1 * 3600$ " has " 1 " and " 3600 " as subterms); solid lines correspond to the direct subterm relation
 231 between a ϕ term $\phi(e_1, e_2)$ and its subterms e_1 and e_2 (they can also be seen as representing the
 232 definition of a symbolic variable as a ϕ term).

233 Switching from terms to cyclic terms and using symbolic variables to name ϕ terms solves *both*
 234 the nondeterminism and termination issues. The names differentiate two terms built from different
 235 nondeterministic choices between two values: e.g. if we have both $x_{\ell_5} = \phi(0, 1)$ and $y_{\ell_6} = \phi(0, 1)$,
 236 we can conclude that $x_{\ell_5} - x_{\ell_5} = 0$ but that $x_{\ell_5} - y_{\ell_6}$ may be different from 0. Furthermore, cyclic
 237 terms provide a finite representation of the values that exist in the program, which is necessary for
 238 the analysis to terminate.

239 Regarding termination, a standard approach consisting of widening [Cousot and Cousot 1977]
 240 two terms to create a cyclic term (e.g., " $0 \nabla 0 + 1$ " = "let $x = \phi(0, x + 1)$ in x ") would be difficult to
 241 implement. Instead, our idea is to view the program locations as the recursion variables in a cyclic
 242 structure, which is the control flow graph of the original program, and to make use of these names
 243 to help us terminate the analysis. This is the reason why our symbolic variables have a program
 244 location in their definition. Essentially, our analysis terminates if variables are named according to
 245



265 Fig. 2. Top: Execution trace of the SSA program of Figure 1c. Middle: The symbolic expression abstraction $p^\#$
 266 seen as an homomorphism or bisimulation. Bottom: Execution trace of the original program of Figure 1b.

267

268

269 a "deterministic" naming scheme based on the program locations: in the actual implementation, we
 270 use integers instead of a set of program variables like `time_i` to name our symbolic variables. We
 271 prove that this technique is sound, but the soundness proofs are unusual.

272 If the global value graph of Figure 1d can be computed explicitly in the analysis, it is not necessary
 273 to do it: computing $p^\#$ (that never contains any ϕ term) is sufficient. Still, the value graph is useful to
 274 justify how $p^\#$ is built, in particular, how the transfer function chooses the name of fresh symbolic
 275 variables and how this choice allows the analysis to terminate.

276 2.2 The SSA graph abstract domain

277

278 2.2.1 *The SSA graph abstraction.* Compared to $p^\#$, the value graph has more information as it links
 279 symbolic variables to their ϕ definition. But the value graph does not capture all the information
 280 about the initial program. First, it misses some relations between symbolic variables: for instance,
 281 we have $\text{hours}_{\ell_3} = \text{time_i}_{\ell_3} / (1 * 3600)$, but this cannot be inferred from the global value graph as
 282 the definition of hours_{ℓ_3} and time_i_{ℓ_3} are on different cycles. Second, the conditions in the original
 283 program appear nowhere in the global value graph.

284 What the global value graph lacks is information about control flow, which is necessary to
 285 encode the fact that different symbolic variables are updated simultaneously, and only when
 286 some conditions are met. Our SSA graph abstraction is built upon the idea of adding control flow
 287 information to the global value graph. More precisely, an SSA graph is a graph whose nodes are
 288 the locations of the original program, and whose edges are associated with a guard (translated
 289 from the guard of the original program) and a set of *bindings* from symbolic variables to symbolic
 290 expressions (corresponding to ϕ nodes in usual SSA). Figure 1c presents the SSA graph computed
 291 by our SSA abstract domain on the program of Figure 1b. Similarly to the global value graph, the
 292 SSA graph links symbolic variables to their ϕ definition; but unlike in the global value graph, it
 293 relates this information to the edges of the control flow. For instance, while the global value graph
 294

only says that time_i_{ℓ_3} is a nondeterministic choice between 0 and time_i_{ℓ_1} , the SSA graph also says that time_i_{ℓ_3} is 0 at ℓ_3 when control comes from ℓ_2 , and then hours_{ℓ_3} will also be 0; that when control comes from ℓ_1 , time_i_{ℓ_3} gets the same value that time_i_{ℓ_1} had, and that this transition is possible only if $\text{time_i}_{\ell_1}/(1 * 3600) < 24$.

The SSA graph can be given a simple semantics as a transition system: states consists in a location ℓ (the same as in the original program) and a *valuation* Γ (mapping from symbolic variables to values). The transition from a state (ℓ, Γ) to a state (ℓ', Γ') is possible only if there is an edge from ℓ to ℓ' in the SSA graph; the guard on this edge evaluates to true; and Γ' is updated from Γ using the bindings on the edge. Finally, a symbolic variable x_ℓ has a scope corresponding to all the locations dominated by ℓ , and variables of a valuation are unbound when they go out of scope. The top of Figure 2 provides an example of an execution trace for the SSA graph of Figure 1c.

2.2.2 A sound and complete abstract interpretation. One of the main result of this paper is that the behavior of the SSA graph computed using abstract interpretation is equivalent to the behavior of the original program. More precisely, the transition systems described by the original program and SSA graph are, once limited to the reachable states, related by a surjective strong homomorphism, a relation that is stronger than bisimulation [Sangiorgi 2009]. Furthermore, this surjective strong homomorphism directly derives from the symbolic expression abstraction $p^\#$: in essence, $p^\#$ is a map from SSA states to program states. Consider Figure 2 again: observe that the abstract stores in the middle can be used to retrieve values for program variables in the states at the bottom from the values of symbolic variables in the SSA states at the top. Thus, to each SSA execution trace corresponds a concrete execution trace (and conversely, every concrete execution traces can be simulated by one SSA execution trace).

This correspondence makes our abstract interpretation both sound (it does not miss any behavior) and complete (it does not include extra behavior). Furthermore, we can make the abstract interpretation complete even when the fixpoint is not finished (see Section 6).

2.2.3 Translation to the usual SSA representation. Our SSA graph representation is a faithful representation of the semantics of the original program, but it abstracts away some details of the usual SSA representation. Thus, translating from an SSA graph to standard SSA consists in additionally constraining the code to fill in these details. For instance, Figure 1e corresponds to a translation of the SSA graph of Figure 1c to a usual representation. The translation has to:

- Name every intermediate computation (i.e., flatten [Ariola and Klop 1996]) the term graph, as SSA representations are usually in three-address code. For instance, our representation names intermediate results like tmp_1 .
- Determine the location where each intermediate computation is done. In Section 5.1 we define a notion of scope of symbolic expression that determines a suitable location for each intermediate computation (although other locations might be preferred). For instance, the assignment $\text{tmp}_1 := \text{one} * \text{big}$ has been moved outside the loop, because the scope $1 * 3600$ is ℓ_0 : i.e., SSA graphs naturally perform loop-invariant code motion.
- Explicit the order of evaluation of symbolic expressions by appropriately scheduling instructions. For instance in Figure 1c the order of assignment between one and big is arbitrary, but must be done before tmp_1 is assigned.

The SSA graph that we present is more appropriate for static analysis. It is however easy to modify SSA graphs so that it is more adapted to compilation, e.g. by using terminator instructions instead of guards on edges, by treating constant expressions specially, or by explicitly restricting the scope of symbolic expressions. This is what our experiment on compilation to LLVM (Section 7.1) does.

$x \in \mathbb{X}$ (program variables)	$z \in \mathbb{Z}$ (numeric constant)	$\diamond \in \{+, -, *, /\}$ (binary operation)
$e \in \mathbb{E}_{\mathbb{X}}$ (program expression) $\triangleq z \mid x \mid e \diamond e$		
$i \in \mathbb{I}$ (instruction) $\triangleq x := e \mid i ; i$		
$\ell \in \mathbb{L}$ (location)	$\mathcal{G} \in \mathbb{G} \triangleq \mathbb{L} \times \mathbb{L} \rightarrow \mathbb{E}_{\mathbb{X}}$ (program graph)	$p \in \mathbb{P} \triangleq (\mathbb{L} \rightarrow \mathbb{I}) \times \mathbb{G}$ (program)
<hr/>		
$\Sigma \triangleq \mathbb{X} \rightarrow \mathbb{Z}$ (stores)	$\mathbb{S} \triangleq \mathbb{L} \times \Sigma$ (states)	$\mathbb{S}_0 \triangleq \{\ell_0\} \times \Sigma$ (initial states)
$\mathcal{E}\cdot : \mathbb{E}_{\mathbb{X}} \times \Sigma \rightarrow \mathbb{Z}$	$\mathcal{E}[z](\sigma) = z$	$\mathcal{E}[x](\sigma) = \sigma[x]$
$\mathcal{E}[e_1 \diamond e_2](\sigma) = \mathcal{E}[e_1](\sigma) \diamond \mathcal{E}[e_2](\sigma)$		
$\mathcal{I}\cdot : \mathbb{I} \times \Sigma \rightarrow \Sigma$	$\mathcal{I}[x := e](\sigma) = \sigma[x \leftarrow \mathcal{E}[e](\sigma)]$	$\mathcal{I}[i_1 ; i_2](\sigma) = \mathcal{I}[i_2](\mathcal{I}[i_1](\sigma))$
$\mapsto \in \mathbb{S} \times \mathbb{S} \quad (\ell_1, \sigma_1) \mapsto (\ell_2, \sigma_2) \triangleq \sigma_2 = \mathcal{I}[\text{instr}[\ell_1]](\sigma_1) \wedge \exists e : ((\ell_1, \ell_2) \mapsto e) \in \mathcal{G} \wedge \mathcal{E}[e](\sigma_2) \neq 0$		

Fig. 3. Syntax (top) and semantics (bottom) of the example language.

3 BACKGROUND AND NOTATIONS

Notations. We view (partial) functions as sets of bindings $x \mapsto y$, replacing the curly braces $\{\}$ of set notation by square brackets $[\]$, e.g., $[x \mapsto x + 1 \mid x \in \mathbb{Z}]$ defines the successor function while $[0 \mapsto 5, 1 \mapsto 7]$ represents an array of two elements. We write $[x \in \mathbb{S} \mapsto y]$ as a shorthand for $[x \mapsto y \mid x \in \mathbb{S}]$. We note the domain of a partial function f by $\text{dom } f$. We note by $A \rightarrow B$ the set of partial functions whose domain is contained in A and codomain is contained in B . The notation $A \twoheadrightarrow B$ denotes the set of total functions from A to B , i.e. partial functions whose domain is exactly A and whose codomain is a subset of B . If f is a function and $a \in \text{dom } f$, we note by $f[a]$ the image of a by f . We define $f[x \leftarrow y] \triangleq [a \in \text{dom } f \mapsto f[a] \mid a \neq x] \cup [x \mapsto y]$ the function f where the binding $x \mapsto y$ has been added (x may or may not be bound in f).

We represent graphs by partial functions from pairs of nodes (representing an edge from the first node to the second) to the label of the edge. Given a graph g and node n , we note $\text{preds}(n) \triangleq \{m : (m, n) \in \text{dom } g\}$ the set of *predecessors* of n in g .

Syntax. We express our analyses on the language defined in Figure 3, featuring assignments of expressions to variables, separation of conditionals into guards and nondeterminism [Dijkstra 1975], and unstructured control flow. It is close to the standard control-flow graph representation of imperative programs.

A program $p \in \mathbb{P} = (\text{instr}[\cdot], \mathcal{G})$ is expressed as a flow-chart doing transitions between *locations* $\ell \in \mathbb{L}$. To each location is mapped an *instruction* $\text{instr}[\ell] \in \mathbb{I}$, consisting in a sequence of assignments. The expressions $e \in \mathbb{E}_{\mathbb{X}}$ are standard (being either variables, constants, or binary operations); $\mathbb{E}_{\mathbb{X}}$ stands for "expressions over the program variables \mathbb{X} ". The possible transitions between locations are expressed using a graph $\mathcal{G} \in (\mathbb{L} \times \mathbb{L}) \rightarrow \mathbb{E}_{\mathbb{X}}$, mapping edges to a *guard* filtering the possible transitions. The choice between multiple outgoing edges is nondeterministic.

We assume the existence of an *initial location* ℓ_0 that has no predecessors. To simplify our definitions and proofs, we assume that there are at most 2 predecessors to any location ℓ .

Concrete semantics. We denote by $\Sigma \triangleq \mathbb{X} \rightarrow \mathbb{Z}$ the set of *stores*, i.e. mapping from variables to values, in the program, and by $\mathbb{S} \triangleq \mathbb{L} \times \Sigma$ the set of *states*. The semantics is given as a transition system (\mathbb{S}, \mapsto) , where \mapsto is a relation describing the transition between states. $\mathbb{S}_0 \triangleq \{\ell_0\} \times \Sigma$ is the set of initial states. The full definition of \mapsto is given in Figure 3, and makes use of $\mathcal{E}\cdot$ and $\mathcal{I}\cdot$ (describing respectively the evaluation of expressions and instructions).

$x_\ell \in \mathbb{V} \triangleq \mathbb{X} \times \mathbb{L}$ (symbolic variables)	$e \in \mathbb{E}_\mathbb{V} \triangleq z \mid x_\ell \mid e \diamond e$ (symbolic expressions)
$\sigma^\# \in \Sigma^\# \triangleq \mathbb{X} \rightarrow \mathbb{E}_\mathbb{V}$ (abstract stores)	$\Sigma_\perp^\# \triangleq \Sigma^\# \cup \{\perp\}$ $\Gamma \in \mathbb{T} \triangleq \mathbb{V} \rightarrow \mathbb{Z}$ (valuations)
$\gamma_{\mathbb{E}_\mathbb{V}} : \mathbb{E}_\mathbb{V} \rightarrow (\mathbb{T} \rightarrow \mathbb{Z})$	$\gamma_{\mathbb{E}_\mathbb{V}}(e) \triangleq [\Gamma \in \mathbb{T} \mapsto \mathcal{E}[e](\Gamma)]$
$\gamma_{\Sigma^\#} : \Sigma^\# \rightarrow (\mathbb{T} \rightarrow \Sigma)$	$\gamma_{\Sigma^\#}(\sigma^\#) \triangleq [\Gamma \in \mathbb{T} \mapsto [x \in \mathbb{X} \mapsto \gamma_{\mathbb{E}_\mathbb{V}}(\sigma^\#[x])(\Gamma)]]$
$\gamma_{\mathbb{L} \rightarrow \Sigma_\perp^\#} : (\mathbb{L} \rightarrow \Sigma_\perp^\#) \rightarrow \mathcal{P}(\mathbb{S})$	$\gamma_{\mathbb{L} \rightarrow \Sigma_\perp^\#}(p^\#) \triangleq \{(\ell, f[\Gamma]) \mid \ell \in \mathbb{L}, p^\#[\ell] \neq \perp, f \in \gamma_{\Sigma^\#}(p^\#[\ell]), \Gamma \in \mathbb{T}\}$
$\mathcal{E}^\#\cdot : \mathbb{E}_\mathbb{X} \times \Sigma^\# \rightarrow \mathbb{E}_\mathbb{V}$	$\mathcal{E}^\#[e](\sigma^\#) \triangleq \text{subst}(e, \sigma^\#)$
$\mathcal{I}^\#\cdot : \mathbb{I} \times \Sigma^\# \rightarrow \Sigma^\#$	$\mathcal{I}^\#[i_1; i_2](\sigma^\#) \triangleq \mathcal{I}^\#[i_2](\mathcal{I}^\#[i_1](\sigma^\#))$
$\phi^{x,\ell} : \mathbb{E}_\mathbb{V} \times \mathbb{E}_\mathbb{V} \rightarrow \mathbb{E}_\mathbb{V}$	$\mathcal{I}^\#[x := e](\sigma^\#) \triangleq \sigma^\#[x \leftarrow \mathcal{E}^\#[e](\sigma^\#)]$
$\sqcup^\ell : \Sigma^\# \times \Sigma^\# \rightarrow \Sigma^\#$	$\phi^{x,\ell}(e_1, e_2) \triangleq \begin{cases} e_1 & \text{if } e_1 = e_2 \\ x_\ell & \text{otherwise} \end{cases}$
$\sigma_1^\# \sqcup^\ell \sigma_2^\# \triangleq [x \in \mathbb{X} \mapsto \phi^{x,\ell}(\sigma_1^\#[x], \sigma_2^\#[x])]$	
$\mathcal{F}_{\mathbb{L} \rightarrow \Sigma_\perp^\#}(p^\#) \triangleq \left[\ell' \in \mathbb{L} \mapsto \begin{cases} [x \in \mathbb{X} \mapsto x_{\ell_0}] & \text{if } \ell' = \ell_0 \\ \sqcup^{\ell'} \{ \mathcal{I}^\#[\text{instr}[\ell]](p^\#[\ell]) \mid \mathcal{G}[\ell, \ell'] \neq 0 \wedge p^\#[\ell] \neq \perp \} & \text{if } \ell' \neq \ell_0 \end{cases} \right]$	

Fig. 4. The symbolic expression domain: abstraction (top), concretizations (middle), and operations (bottom).

Least fixpoint and collecting semantics. Given $\mathcal{F}_S \in S \rightarrow S$, we note by $\text{lfp}(\mathcal{F}_S)$ the *least fixpoint* of \mathcal{F}_S , i.e. the smallest element $X \in S$ such that $\mathcal{F}_S(X) = X$. We will mainly use the set of reachable states as our main collecting semantics. It is expressed as the least fixpoint of the following function $\mathcal{F}_{\mathcal{P}(\mathbb{S})} \in \mathcal{P}(\mathbb{S}) \rightarrow \mathcal{P}(\mathbb{S})$:

$$\mathcal{F}_{\mathcal{P}(\mathbb{S})}(X) \triangleq \mathbb{S}_0 \cup \{s' \mid \exists s \in X : s \mapsto s'\}$$

The set of reachable states is noted by \mathbb{S}^* , i.e.

$$\mathbb{S}^* = \text{lfp}(\mathcal{F}_{\mathcal{P}(\mathbb{S})}) = \{s \in \mathbb{S} : \exists s_0 \in \mathbb{S}_0, s_0 \mapsto^* s\}$$

where \mapsto^* is the transitive and reflexive closure of the transition relation \mapsto .

4 SYMBOLIC EXPRESSION ANALYSIS

This section formalizes the definition of the symbolic expression analysis presented in the overview (Section 2.1). The analysis is formalized as a forward abstract interpretation [Cousot and Cousot 1977].

4.1 The symbolic expression abstract domain

4.1.1 Symbolic expression, abstract store, and main program abstraction. The structure of our abstraction (fully defined in Figure 4) follows the concrete semantics.

Concrete values are abstracted by *symbolic expressions* $e \in \mathbb{E}_\mathbb{V}$, where $\mathbb{E}_\mathbb{V}$ is similar to expressions $\mathbb{E}_\mathbb{X}$ except that variables are not program variables \mathbb{X} , but elements from a set \mathbb{V} of *symbolic variables*. In this section, $\mathbb{V} \triangleq \mathbb{X} \times \mathbb{L}$, and we note symbolic variables by x_ℓ where $x \in \mathbb{X}$ and $\ell \in \mathbb{L}$. Note that the fact that our symbolic variables are given a deterministic, unique name (instead of generating "fresh" symbolic variables as is common in existing formula-based abstract domains (e.g., [Chang

and Leino 2005; Chang and Rival 2013; Gange et al. 2016; Illous et al. 2021]) is a distinctive feature of our analysis.

As is common in abstract interpretation [Cousot and Cousot 1977], symbolic expressions are given a meaning using a concretization function $\gamma_{\mathbb{E}_V}$, that associates an abstract element to the set of concrete elements that it describes. Instead of concretizing a symbolic expression to a set of values, we concretize it as a function from valuations to values, where a valuation Γ is a function that defines how symbolic variables are bound to a concrete value. Given a symbolic expression e , $\gamma_{\mathbb{E}_V}(e)$ is a function that takes a Γ and returns the value obtained by evaluation of the symbolic expression after replacement of the symbolic variables by their value in Γ . This function can be seen as representing a set of values (the image of the function); but the point of having a function is that different symbolic expressions can be related by sharing a Γ .

Concrete stores are abstracted by *abstract stores* $\sigma^\# \in \Sigma^\# \triangleq \mathcal{X} \rightarrow \mathbb{E}_V$, which simply maps program variables to symbolic expressions. Sometimes we need to represent a set of no abstract store, and use the symbol \perp for this; we define $\Sigma_\perp^\# \triangleq \Sigma^\# \cup \{\perp\}$. The concretization function for abstract stores $\gamma_{\Sigma^\#}$ is a function from valuations to concrete stores; here we observe that the same Γ is passed to the concretization of all the symbolic expression in the store to get the corresponding value, thereby allowing relations between symbolic expressions. For instance, the abstract store $[x \mapsto 2 * z_{\ell_1}; y \mapsto 6 * z_{\ell_1}; z \mapsto z_{\ell_3}]$ represents concrete stores where x is even, y is a multiple of x , and z is arbitrary. Again, the function returned by $\gamma_{\Sigma^\#}$ can be seen as set of concrete stores (the image of the function).

Finally, as is standard in flow-sensitive dataflow analyses, our main program abstraction is an element $p^\# \in \mathbb{L} \rightarrow \Sigma_\perp^\#$ assigning to each program location an abstract store, or \perp (which happens when the location is unreachable, or the computation of the program abstraction is incomplete). We concretize elements $p^\# \in \mathbb{L} \rightarrow \Sigma_\perp^\#$ as a set of states: a state (ℓ, σ) is described by $p^\#$ whenever σ is described by the abstract store $p^\#[\ell]$ (i.e., if Γ exists such that $\gamma_{\Sigma^\#}(p^\#[\ell])(\Gamma) = \sigma$).

4.1.2 Analysis operations. The abstract evaluation of expressions ($\mathcal{E}^\#\cdot$) consists in substituting the free program variables in the expression with the corresponding symbolic expression in the abstract store. The abstract evaluation of instructions ($\mathcal{I}^\#\cdot$) mimics the concrete evaluation by updating the abstract store after abstract evaluation of the expression.

The most interesting operation is the join operation \sqcup^ℓ , which rely on a kind of "join of expressions" performed by a function ϕ . The current definition of ϕ is simple: when the two arguments of ϕ differ, a new symbolic variable is created; when they are equal, that expression is returned (this latter case corresponds to the notion of minimal number of ϕ functions in an SSA translation [Cytron et al. 1991]). The additional x, ℓ argument allows deterministic creation of variable names, which is important to ensure termination of the analysis. The \sqcup^ℓ operation is extended to operate on a set (i.e. $\sqcup^\ell\{\cdot\} = \perp$, $\sqcup^\ell\{\sigma^\#\} = \sigma^\#$, $\sqcup^\ell\{\sigma_1^\#, \sigma_2^\#\} = \sigma_1^\# \sqcup^\ell \sigma_2^\#$).

Example 4.1. When analyzing ℓ_1 after the first loop iteration, we have to join two abstract stores:

$$\begin{aligned} \sigma_1^\# &= [\text{inc} \mapsto 1, \text{time} \mapsto 0, i \mapsto 0, \text{hours} \mapsto \text{hours}_{\ell_0}] \text{ and} \\ \sigma_2^\# &= [\text{inc} \mapsto 1, \text{time} \mapsto 0 + 1, i \mapsto 0 + 1, \text{hours} \mapsto \text{hours}_{\ell_3}]. \end{aligned}$$

The result is in Figure 1a (top): `inc` is bound to 1 in both incoming abstract stores, and thus stays bound to 1; while a new symbolic variable is created for `time`, `i` and `hours`.

The main transfer function $\mathcal{F}_{\mathbb{L} \rightarrow \Sigma_\perp^\#}$ does a classic fixpoint step in flow-sensitive dataflow analysis, by merging at each location ℓ' the states coming from its predecessors. Note that this transfer function makes use of the information that a predecessor location ℓ is unreachable (i.e., $p^\#[\ell] = \perp$)

491 or that the guard on an edge always evaluates to false (i.e. $\mathcal{G}[\ell, \ell'] = 0$) to improve its precision.
 492 When combined with other domains (e.g. using a reduced product [Cousot and Cousot 1979] with
 493 a numerical abstraction), it is easy for the symbolic expression domain to benefit from interaction
 494 with the other domains by determining more unreachable locations or dead branches; in the end this
 495 mechanism ensures the automatic preservation of SSA invariants when another domain discovers
 496 dead code.

497 There is an order on $\mathbb{L} \rightarrow \Sigma_{\perp}^{\#}$ (see Section 6) and the lattice has a finite height (which is $|\mathbb{L}| \times |\mathcal{X}|$,
 498 where $|S|$ is the cardinal of S): hence, a Kleene iteration [Cousot and Cousot 1977] of $\mathcal{F}_{\mathbb{L} \rightarrow \Sigma_{\perp}^{\#}}$ will
 499 find a fixpoint in a finite number of steps.

500
 501 **4.1.3 Soundness of the analysis.** Due to space constraints, the full proof of soundness of this
 502 analysis is put in Appendix (Appendix A). Here we summarize the main interesting points about
 503 the proof.

504 First, note that if the SSA abstraction is sound and complete, the symbolic expression analysis is
 505 sound, but incomplete. It is in fact an advantage to have an incomplete abstraction that goes along
 506 the exact SSA abstraction: while the SSA abstraction exactly translates the semantics of the original
 507 program, the concretization of the incomplete the symbolic expression domain allows assessing
 508 the precision of this translation.

509 That said, a second observation is that the $\mathcal{E}^{\#}\cdot$ and $\mathcal{I}^{\#}\cdot$ operations are exact, in addition
 510 to being sound. This property is important to prove the soundness and completeness of the SSA
 511 translation.

512 A last, unusual, point is that the \sqcup^{ℓ} operator *is not sound* when applied to any pair of elements in
 513 $\Sigma^{\#}$. The problem comes from the fact that we reuse variable identifiers, which could thus create
 514 spurious relations between terms (this problem does not happen in analyses which only create
 515 fresh variables). Consider for instance:

$$516 \quad [x \mapsto 1 + y_{\ell_0}, y \mapsto 2] \sqcup^{\ell_0} [x \mapsto 1 + y_{\ell_0}, y \mapsto 3] = [x \mapsto 1 + y_{\ell_0}, y \mapsto y_{\ell_0}]$$

517
 518 The resulting abstract store implies the relation $x = 1 + y$ through the shared use of y_{ℓ_0} , but this
 519 relation is false in both arguments to \sqcup^{ℓ_0} .

520 However, the complete analysis is sound, because 1. \sqcup^{ℓ} is sound in the case when ℓ does not
 521 occur in at least one of the arguments to \sqcup^{ℓ} , and 2. during the fixpoint computation, $p^{\#}$ is always
 522 such that \sqcup^{ℓ} is applied in the sound case (because ℓ never occurs in the abstract stores on the simple
 523 paths from ℓ_0 to ℓ).
 524
 525

526 4.2 Building ϕ dynamically

527 The abstract domain defined Figure 4 sometimes assigns different symbolic expressions to program
 528 variables that could be proved equal. Consider again Example 4.1 (page 10). The program variables
 529 `time` and `i` are bound to the same symbolic expression in each of $\sigma_1^{\#}$ and $\sigma_2^{\#}$, and are thus equal;
 530 it would thus be more precise to have these program variables be bound to the same symbolic
 531 variable. What we want is in Figure 1a ($p^{\#}[\ell_1]$ on the second line): there, both `time` and `i` are bound
 532 to the same symbolic variable `time_i_{\ell_1}`, which represents the value of both `time` and `i` at ℓ_1 .
 533
 534

535 *Building ϕ dynamically.* For this to work, we need to perform small changes to our static analysis.
 536 The ϕ function can no longer have a static definition, but will be computed based on the arguments
 537 of \sqcup^{ℓ} . For this, we define $\Phi \triangleq ((\mathbb{E}_{\vee} \times \mathbb{E}_{\vee}) \rightarrow \mathbb{E}_{\vee})$ to be the set of ϕ functions, and we define
 538 $\Phi^{\ell} : \Sigma^{\#} \times \Sigma^{\#} \rightarrow \Phi$ as the function computing ϕ from two abstract states. A definition for Φ^{ℓ} that
 539

returns a ϕ equivalent to the one statically defined in Figure 4 is as follows:

$$\begin{aligned}
\Phi_{=}^{\ell} &: \Sigma^{\#} \times \Sigma^{\#} \rightarrow \Phi & \Phi_{=}^{\ell}(\sigma_1^{\#}, \sigma_2^{\#}) &\triangleq [(e, e) \mapsto e \mid \exists x \in \mathbb{X} : \sigma_1^{\#}[x] = \sigma_2^{\#}[x] = e] \\
\Phi_{\neq}^{\ell} &: \Sigma^{\#} \times \Sigma^{\#} \rightarrow \Phi & \Phi_{\neq}^{\ell}(\sigma_1^{\#}, \sigma_2^{\#}) &\triangleq [(\sigma_1^{\#}[x], \sigma_2^{\#}[x]) \mapsto x_{\ell} \mid x \in \mathbb{X} \wedge \sigma_1^{\#}[x] \neq \sigma_2^{\#}[x]] \\
\Phi^{\ell} &: \Sigma^{\#} \times \Sigma^{\#} \rightarrow \Phi & \Phi^{\ell}(\sigma_1^{\#}, \sigma_2^{\#}) &\triangleq \Phi_{\neq}^{\ell}(\sigma_1^{\#}, \sigma_2^{\#}) \cup \Phi_{=}^{\ell}(\sigma_1^{\#}, \sigma_2^{\#})
\end{aligned}$$

We actually defined three Φ functions: $\Phi_{=}^{\ell}$ returns the ϕ function for the cases where the arguments to ϕ are equal, Φ_{\neq}^{ℓ} for the cases where they are different, and Φ^{ℓ} handles both cases. Later on, we will focus on Φ_{\neq}^{ℓ} as it corresponds to the ϕ functions in standard SSA.

We now need to change the definition of \sqcup^{ℓ} to make use of Φ^{ℓ} :

$$\sigma_1^{\#} \sqcup^{\ell} \sigma_2^{\#} \triangleq \text{let } \phi \triangleq \Phi^{\ell}(\sigma_1^{\#}, \sigma_2^{\#}) \text{ in } [x \in \mathbb{X} \mapsto \phi(\sigma_1^{\#}[x], \sigma_2^{\#}[x])]$$

More precise global value numbering. With the above changes, we compute the same analysis as in the previous section, the only difference being that the ϕ function is computed dynamically. But we can now adapt the analysis to return the same symbolic variable when two pairs of variables point to the same symbolic expression. It suffices first to change the definition of \mathbb{V} so that symbolic variables are named according to a set of program variables $X \in \mathcal{P}(\mathbb{X})$, instead of a single variable $x \in \mathbb{X}$: i.e., now $\mathbb{V} \triangleq \mathcal{P}(\mathbb{X}) \times \mathbb{L}$. Second, we change the definition of Φ_{\neq}^{ℓ} as follows:

$$\Phi_{\neq}^{\ell}(\sigma_1^{\#}, \sigma_2^{\#}) \triangleq [(e_1, e_2) \mapsto X_{\ell} \mid e_1 \neq e_2 \wedge X \neq \{\} \wedge \forall x \in X : \sigma_1^{\#}[x] = e_1 \wedge \sigma_2^{\#}[x] = e_2]$$

This new definition associates to each pair of different expressions e_1, e_2 all the variables that contain these expressions in the abstract stores. The new analysis is still sound, and all the theorems of Section A still hold (only the proof of Lemma A.6 needs to be modified).

Note. The formalization requires \sqcup^{ℓ} to perform two passes (one to build ϕ , and one to use it), but it is feasible to do it in one pass. It suffices to return a fresh (integer) identifier every time a new pair of expressions e_1, e_2 is passed to ϕ , and modify, using a mutable state, the relation between the identifier and the set of variables that it represents. Actually the implementation may only use the fresh identifier and completely omit to represent the set of variables, which is useful only to understand what the integer identifier (i.e., value number [Rosen et al. 1988]) represents.

Building the global value graph. We return to the ϕ function computed by $\Phi_{\neq}^{\ell_1}$ (Example 4.1), which is

$$\Phi_{\neq}^{\ell_1}(\sigma_1^{\#}, \sigma_2^{\#}) = [(0, \text{time_}i_{\ell_3}) \mapsto \text{time_}i_{\ell_1}, (\text{hours}_{\ell_0}, \text{hours}_{\ell_3}) \mapsto \text{hours}_{\ell_1}]$$

In other words, $\text{time_}i_{\ell_1} = \phi(0, \text{time_}i_{\ell_3})$ and $\text{hours}_{\ell_1} = \phi(\text{hours}_{\ell_0}, \text{hours}_{\ell_3})$: the computed ϕ function corresponds to the binding of a symbolic variable to a ϕ expression in usual SSA. Thus, if we first collect all the terms that appear in $p^{\#}$, the result of the symbolic analysis, with all the ϕ functions computed by $\mathcal{F}_{\mathbb{L} \rightarrow \Sigma_1^{\#}}(p^{\#})$ viewed as a definition of symbolic variables, we obtain a cyclic term graph which is the global value graph. This can be done efficiently and incrementally; we do not explain this further as the generation of SSA is a generalization of this construction (as SSA is the global value graph with additional information about the control-flow).

4.3 Note on practical implementation

Several points must be observed for an efficient practical implementation. Firstly, the symbolic expressions should be hash-consed (i.e. the terms should be built as a minimal DAG [Reif and Lewis 1986]). This ensures that the total memory taken by the symbolic expressions corresponds to the number of nodes in the global value graph, and that the complexity of a $\mathcal{E}^{\#}[\![e]\!](\sigma^{\#})$ operation is in $O(|e|)$ (i.e. proportional to the number of nodes in e).

Secondly, the abstract stores $\sigma^\# \in \Sigma^\#$ should be implemented using balanced binary search trees [Blanchet et al. 2002] or, better, Okasaki maps [Okasaki and Gill 1998] (see Section 7.1). This allows the complexity of insertion in the map for the interpretation of assignments to be in $O(\log |\mathbb{X}|)$, and maximum complexity of the \sqcup^ℓ operation to be in $O(|\mathbb{X}|)$ (but the average complexity of the \sqcup^ℓ operation is in $O(\Delta \log |\mathbb{X}|)$, where Δ is the number of variables that differs in the arguments of \sqcup^ℓ).

Thirdly, a chaotic iteration strategy [Cousot 1977, 1978] should be used to propagate states efficiently. For instance, using the iterative iteration strategy of Bourdoncle [1993] ensures that each location of the program will be analyzed less than h time, where h is the height of the lattice of abstract stores. It can be shown that $h = |\mathbb{X}|$ (we can view $\mathbb{E}_\mathbb{V}$ as a lattice of height 2, and abstract stores contain $|\mathbb{X}|$ of them).

Thus, assuming that the size of all expressions in the program is bound by a small constant (in addition to the assumption that nodes have no more than 2 predecessors), the maximum complexity of our analysis is $O(|\mathbb{L}| * |\mathbb{X}|^2)$. However, in a typical analysis, only a few iterations of the outermost component will be needed for the fixpoint iteration to converge (in general almost all the program variables converge simultaneously), the number of join points is smaller than the number of instructions, and many program variables have the same expression during joins (especially if there are many variables), so the typical complexity is $O(|\mathbb{L}| * \log |\mathbb{X}|)$, which makes the analysis quite efficient in practice (see Section 7.1).

Finally, the evaluation of expressions may perform more than just substitutions – it can perform some rewriting operations such as arithmetic simplifications, constant folding, normalization of commutative operations, etc. so as to detect more equalities between variables. It should also be possible to replace symbolic expressions by equivalent classes of equal expressions – using equality saturation [Nelson 1980; Willsey et al. 2021] to find more equalities between expressions.

5 CONSTRUCTION OF AN SSA GRAPH USING ABSTRACT INTERPRETATION

The previous section explained how a symbolic expression abstraction $p^\# \in \mathbb{L} \rightarrow \Sigma_\perp^\#$ could be computed using an analysis based on abstract interpretation. This section explains how this abstraction can be turned into an SSA graph.

5.1 Scope and dominance properties

First, we study some properties of the result $p^\# \in \mathbb{L} \rightarrow \Sigma_\perp^\#$ of a symbolic expression analysis. More generally, in this section, we assume that $p^\#$ is a fixpoint of $\mathcal{F}_{\mathbb{L} \rightarrow \Sigma_\perp^\#}$ (although the results also apply when $p^\#$ is a postfixpoint) in a program whose graph is \mathcal{G} . The study of these properties is important to define the notion of *scope* and the semantics of SSA graphs.

We recall the notion of *domination*: we say that ℓ_D *dominates* ℓ (in a graph \mathcal{G}) if every path from ℓ_0 to ℓ in the graph \mathcal{G} must go through ℓ_D . We also define the *location of a variable*: if $v \in \mathbb{V}$ is a pair whose second element is a location ℓ , then $\text{loc}(v) = \ell$. Informally, $\text{loc}(v)$ corresponds, in traditional SSA, to the basic block where the variable v will be defined.

We saw that the symbolic expression analysis (alone or paired with other abstract domains) can determine if an edge or a location is unreachable. To simplify our definitions, we assume that dead code elimination has been performed on \mathcal{G} – otherwise, excluding the paths that go through unreachable nodes would make some notions, like domination, more complex.

Strictness. First we prove a property corresponding to the *dominance* or *strictness* property of SSA [Rastello 2016]. In strict SSA, if a variable occurs at location ℓ , then it is defined at a location ℓ_D that dominates ℓ . Note however that our theorem apply to a symbolic expression abstraction $p^\# \in \mathbb{L} \rightarrow \Sigma_\perp^\#$, which is a different mathematical object than an SSA program.

THEOREM 5.1 (STRICTNESS). $\forall \ell \in \mathbb{L}, x \in \mathbb{X}, v \in \mathbb{V} : v \text{ occurs in } p^\sharp[\ell][x] \Rightarrow \text{loc}(v) \text{ dominates } \ell$.

See [proof](#) on page 33.

Note that this result only applies when p^\sharp is a fixpoint: during the fixpoint computation, there may be a variable x_{ℓ_D} that gets propagated to a location ℓ without ℓ being dominated by ℓ_D (e.g. when another path to ℓ is not yet propagated).

Scope. We now define a notion of *scope*, that intuitively corresponds to the set of locations on which an expression has a meaning: i.e. the locations that are dominated by the locations of all the variables of the expression.

Definition 5.2 (Scope). Given a graph G , the function $\text{scope}_G : \mathbb{E}_V \rightarrow \mathcal{P}(\mathbb{L})$ is defined as follows:

$$\text{scope}_G(v) \triangleq \{\ell \in \mathbb{L} \mid \text{loc}(v) \text{ dominates } \ell \text{ in the graph } G\} \quad \text{scope}_G(z) \triangleq \mathbb{L}$$

$$\text{scope}_G(e_1 \diamond e_2) \triangleq \text{scope}_G(e_1) \cap \text{scope}_G(e_2)$$

Another interpretation of the scope of an expression e is the set of locations of p^\sharp where e may appear when the analysis is complete.

THEOREM 5.3 (p^\sharp RESPECTS SCOPE). $\forall \ell \in \mathbb{L} : \forall x \in \mathbb{X} : \ell \in \text{scope}_G(p^\sharp[\ell][x])$

PROOF. Direct consequence of Theorem 5.1 and definition of scope. \square

We say that the scope of e is *defined* when $\text{scope}_G(e)$ is non-empty. Intuitively, if the scope of a symbolic expression e is undefined, then e is meaningless.

COROLLARY 5.4. $\forall \ell \in \mathbb{L} : \forall x \in \mathbb{X} : \text{scope}_G(p^\sharp[\ell][x])$ is defined.

Example 5.5. Here is the scope of some expressions for the program of Figure 1b, where \mathcal{G} corresponds to the graph of the program. All these expressions are defined except $\text{time_i}_{\ell_2} + \text{hours}_{\ell_3}$.

$$\begin{aligned} \text{scope}_G(\text{hours}_{\ell_1}) &= \{\ell_1, \ell_2, \ell_3\} & \text{scope}_G(1 * 3600) &= \mathbb{L} = \{\ell_0, \ell_1, \ell_2, \ell_3\} \\ \text{scope}_G(\text{time_i}_{\ell_2}) &= \{\ell_2\} & \text{scope}_G(\text{hours}_{\ell_1} + \text{hours}_{\ell_3}) &= \{\ell_1, \ell_2, \ell_3\} \cap \{\ell_3\} = \{\ell_3\} \\ \text{scope}_G(\text{hours}_{\ell_3}) &= \{\ell_3\} & \text{scope}_G(\text{time_i}_{\ell_2} + \text{hours}_{\ell_3}) &= \{\ell_2\} \cap \{\ell_3\} = \{\} \end{aligned}$$

Single-location scope. The scope of any expression, when defined, can be represented by a single location; this property comes from the fact that domination is a tree relation [Appel 1998a]. The following function $\overline{\text{scope}}$ expresses how this location can be computed (where the min of two locations is the location dominated by the other; this min exists when the scope of e is defined).

$$\overline{\text{scope}}_G(v) \triangleq \text{loc}(v) \quad \overline{\text{scope}}_G(z) \triangleq \ell_0 \quad \overline{\text{scope}}_G(e_1 \diamond e_2) \triangleq \min(\overline{\text{scope}}_G(e_1), \overline{\text{scope}}_G(e_2))$$

THEOREM 5.6. $\text{scope}_G(e) = \{\ell \mid \overline{\text{scope}}_G(e) \text{ dominates } \ell\}$

See [proof](#) on page 33.

We use this $\overline{\text{scope}}$ operation when translating an expression in our SSA representation into assignment of a temporary variable in usual SSA: $\overline{\text{scope}}(e)$ represents the most hoisted location where to place the assignment.

5.2 The SSA graph

We can now present (Figure 5) the syntax and semantics of our variant of SSA, the SSA graph (note that some authors [Stanier 2016] call the global value graph the SSA graph, which should not be confused with the SSA graph that we present here). The main use of this form of SSA is to be used as an abstraction of an abstract domain; however, its simple syntax and semantics may be useful in

$$\begin{array}{l}
\hat{\mathbb{G}} \triangleq (\mathbb{L} \times \mathbb{L}) \rightarrow (\mathbb{E}_{\mathbb{V}} \times (\mathbb{V} \rightarrow \mathbb{E}_{\mathbb{V}})) \text{ (SSA graph)} \quad \hat{\mathbb{S}} \triangleq \mathbb{L} \times \mathbb{T} \text{ (SSA state)} \quad \mathbb{T} \triangleq \mathbb{V} \rightarrow \mathbb{Z} \text{ (valuation)} \\
\hline
\text{unbind}_{\hat{g}} : \mathbb{L} \times \mathbb{T} \rightarrow \mathbb{T} \quad \text{unbind}_{\hat{g}}(\ell, \Gamma) \triangleq [v \in \text{dom } \Gamma \mapsto \Gamma[v] \mid \ell \in \text{scope}_{\hat{g}}(v)] \\
\text{bind} : (\mathbb{V} \rightarrow \mathbb{E}_{\mathbb{V}}) \times \mathbb{T} \rightarrow \mathbb{T} \quad \text{bind}(B, \Gamma) \triangleq \left[v \in \mathbb{V} \mapsto \begin{cases} \mathcal{E}[\llbracket B[v] \rrbracket](\Gamma) & \text{if } v \in \text{dom } B \\ \Gamma[v] & \text{otherwise} \end{cases} \right] \\
\rightsquigarrow_{\hat{g}} \in \hat{\mathbb{S}} \times \hat{\mathbb{S}} \quad (\ell, \Gamma) \rightsquigarrow_{\hat{g}} (\ell', \Gamma') \triangleq \exists e \in \mathbb{E}_{\mathbb{V}}, B \in \mathbb{V} \rightarrow \mathbb{E}_{\mathbb{V}} : \begin{array}{l} (\ell, \ell') \mapsto (e, B) \in \hat{g} \\ \wedge \mathcal{E}[\llbracket e \rrbracket](\Gamma) \neq 0 \\ \wedge \Gamma' = \text{unbind}_{\hat{g}}(\ell', \text{bind}(B, \Gamma)) \end{array}
\end{array}$$

Fig. 5. Syntax and semantics of SSA graphs

other contexts. A distinctive feature of this form (compared to traditional SSA form) is that it is *implicitly scoped*: we define the location of definition only for the symbolic variables (corresponding to the phi variables in traditional SSA form), and not for every expression (corresponding to other variables in traditional three-address SSA form). However, the largest scope in which an expression e is defined is given by $\text{scope}(e)$ in the previous section; hence the scope of e is defined implicitly (and we can use this scope when translating to traditional SSA).

Syntax of SSA graphs. Our SSA representation is a graph $\hat{g} \in \hat{\mathbb{G}}$ with $\hat{\mathbb{G}} \triangleq (\mathbb{L} \times \mathbb{L}) \rightarrow \mathbb{E}_{\mathbb{V}} \times (\mathbb{V} \rightarrow \mathbb{E}_{\mathbb{V}})$. The graph consists in a set of bindings $(\ell, \ell') \mapsto (e, B)$ representing an edge from ℓ to ℓ' labeled by a guard e and binding B , where a binding is a mapping from some variables to terms. Figure 1c presents the SSA graph produced by our analysis of the program of Figure 1b.

Semantics of SSA graphs. A graph $\hat{g} \in \hat{\mathbb{G}}$ represents a transition system $(\hat{\mathbb{S}}, \rightsquigarrow_{\hat{g}})$ between symbolic states $\hat{s} \in \hat{\mathbb{S}} \triangleq \mathbb{L} \times \mathbb{T}$ where a symbolic state consists of a control location ℓ and a valuation. The valuations $\Gamma \in \mathbb{T} \triangleq \mathbb{V} \rightarrow \mathbb{Z}$ are those used in the concretization of our symbolic expression abstraction, and assign values to symbolic variables. The initial states $\hat{\mathbb{S}}_0 \triangleq \{(\ell_0, \Gamma) \mid \text{dom } \Gamma = \{x_{\ell_0} \mid x \in \mathbb{X}\}\}$ intuitively describes the initial value of each program variable at the starting location ℓ_0 .

The transition relation $\rightsquigarrow_{\hat{g}}$, formally defined in Figure 5, works as follows: transition from a state (ℓ, Γ) to a state at location ℓ' in the graph is possible only if 1. there is a $(\ell, \ell') \mapsto (e, B)$ edge in the graph \hat{g} , and 2. the guard e evaluates to true. The transition then consists in adding some new bindings to Γ using the definitions provided by B (operator `bind`), but also to unbind the variables that go out of scope in the new control location ℓ' , using the notion of scope that we defined in the previous section. An example of execution trace for the SSA program given in Figure 1c is given Figure 2 (first line).

The use of `unbind` is unusual in SSA semantics: generally SSA variables are assigned rather than bound (see e.g. [Barthe et al. 2014; Cytron et al. 1991; Zhao et al. 2012]). Our proof of soundness still holds with the assignment semantics (and is actually simpler), but `unbind` is important when SSA graphs are used in a combination with other abstract domains (e.g., to limit the scope where information about a variable must be retained).

Note that the semantics of SSA uses $\text{scope}_{\hat{g}}$ which depends on the SSA graph \hat{g} instead of the program graph \mathcal{G} , as we want the SSA semantics to be independent of the original program. Note however, that $\text{scope}_{\hat{g}}$ and $\text{scope}_{\mathcal{G}}$ are the same when we define $\hat{\mathcal{G}}$ to be the subset of the original program that excludes the parts that the symbolic expression analysis can prove unreachable.

$$\begin{aligned}
\text{graph}(p^\sharp) &\triangleq \bigcup_{(\ell, \ell') \in \text{dom } \mathcal{G}} \text{edge}(p^\sharp, \ell, \ell') \\
\text{edge}(p^\sharp, \ell, \ell') &\triangleq [(\ell, \ell') \mapsto (\hat{e}, B) \mid e \neq 0 \wedge p^\sharp[\ell] \neq \perp] \text{ where} \\
e &= \mathcal{G}[\ell, \ell'], \\
\sigma'^\sharp &= \mathcal{I}^\sharp[\text{instr}[\ell]](p^\sharp[\ell]), \\
\hat{e} &= \mathcal{E}^\sharp[e](\sigma'^\sharp), \\
B &= \{p^\sharp[\ell'][x] \mapsto \sigma'^\sharp[x] \mid x \in \mathbb{X} \wedge p^\sharp[\ell'][x] \neq \sigma'^\sharp[x]\}
\end{aligned}$$

Fig. 6. Translation of a symbolic expression abstraction to an SSA graph

5.3 From symbolic expressions to SSA graphs

We now explain how the symbolic expression abstraction p^\sharp can be translated into an SSA graph (Figure 6). The idea is that each edge of the original program corresponds to at most one edge in the SSA translation : if we detect that the source location ℓ is unreachable (i.e., $p^\sharp[\ell] = \perp$) or the guard is not satisfiable (i.e. $e = 0$), we leave the corresponding edge out of the SSA graph. The translation is done as follows: we first compute the abstract store σ'^\sharp after execution of the block, and use it to translate the original guard e into an SSA guard \hat{e} ; finally we examine the difference between σ'^\sharp and $p^\sharp[\ell']$, the abstract store at ℓ' , so that there is a binding in B whenever a new symbolic variable was introduced.

Example 5.7. Let p^\sharp be the result of the analysis of our running example (Figure 1a). Then, $p^\sharp[\ell_1] \neq \perp$ and $\mathcal{G}[\ell_1, \ell_3] = \text{"hours} < 24" \neq 0$, thus we have an edge from ℓ_1 to ℓ_3 in the SSA graph. Moreover:

$\text{edge}(p^\sharp, \ell_1, \ell_3) = \{(\ell_1, \ell_3) \mapsto (\mathcal{E}^\sharp[e](\sigma'^\sharp), \{p^\sharp[\ell_3][x] \mapsto \sigma'^\sharp[x] \mid x \in \mathbb{X} \wedge p^\sharp[\ell_3][x] \neq \sigma'^\sharp[x]\})\}$ where

$$\sigma'^\sharp = \mathcal{I}^\sharp[\text{instr}[\ell_1]](p^\sharp[\ell_1]) = \begin{bmatrix} \text{inc} & \mapsto 1 \\ \text{time} & \mapsto \text{time_i}_{\ell_1} \\ \text{i} & \mapsto \text{time_i}_{\ell_1} \\ \text{hours} & \mapsto \text{time_i}_{\ell_1} / (1 * 3600) \end{bmatrix} \text{ and } p^\sharp[\ell_3] = \begin{bmatrix} \text{inc} & \mapsto 1 \\ \text{time} & \mapsto \text{time_i}_{\ell_3} \\ \text{i} & \mapsto \text{time_i}_{\ell_3} \\ \text{hours} & \mapsto \text{hours}_{\ell_3} \end{bmatrix}$$

The result is

$$\text{edge}(p^\sharp, \ell_1, \ell_3) = \left\{ (\ell_1, \ell_3) \mapsto \left(\text{time_i}_{\ell_1} / (1 * 3600) < 24, \begin{bmatrix} \text{time_i}_{\ell_3} \mapsto \text{time_i}_{\ell_1} \\ \text{hours}_{\ell_3} \mapsto \text{time_i}_{\ell_1} / (1 * 3600) \end{bmatrix} \right) \right\}$$

as in Figure 1c.

5.4 The SSA abstract domain

5.4.1 Abstract elements. We can now describe our SSA abstract domain (Figure 7). Elements of this domain are a pair (p^\sharp, \hat{g}) where $p^\sharp \in \mathbb{L} \rightarrow \Sigma_\perp^\sharp$ is an element of the symbolic expression abstract domain and $\hat{g} \in \hat{\mathcal{G}}$ is an SSA graph. Note that p^\sharp is a flow-sensitive abstraction while \hat{g} is a flow-insensitive abstraction.

5.4.2 Concretization. The central idea of the concretization is that an element $p^\sharp \in \mathbb{L} \rightarrow \Sigma_\perp^\sharp$ can be viewed as a function $\mathcal{H}_{p^\sharp} \in \hat{\mathcal{S}} \rightarrow \mathcal{S}$ mapping SSA states to concrete states. More precisely, given an SSA state $\hat{s} = (\ell, \Gamma)$ composed of a location ℓ and a valuation Γ , the abstract store $p^\sharp[\ell]$ can be viewed as a function from valuations to concrete stores using the concretization γ_{Σ^\sharp} : indeed, $\gamma_{\Sigma^\sharp}(p^\sharp[\ell]) \in \mathbb{F} \rightarrow \Sigma$. \mathcal{H}_{p^\sharp} retrieves a concrete store by applying Γ to this function, keeps the location ℓ , and use both to create a concrete state.

$$\begin{array}{c}
\text{SSA}^\# = (\mathbb{L} \rightarrow \Sigma_\perp^\#) \times \hat{\mathbb{G}} \\
\hline
\mathcal{H}_{p^\#} : \hat{\mathbb{S}} \rightarrow \mathbb{S} \qquad \gamma_{\hat{\mathbb{G}}}(\hat{g}) : \hat{\mathbb{G}} \rightarrow \mathcal{P}(\mathcal{P}(\hat{\mathbb{S}} \times \hat{\mathbb{S}})) \\
\mathcal{H}_{p^\#}(\ell, \Gamma) \triangleq (\ell, \gamma_{\Sigma^\#}(p^\#[\ell])(\Gamma)) \qquad \gamma_{\hat{\mathbb{G}}}(\hat{g}) \triangleq \rightsquigarrow_{\hat{g}} \hat{\mathbb{S}}^* \\
\gamma_{\text{SSA}^\#} : \text{SSA}^\# \rightarrow \mathcal{P}(\mathcal{P}(\mathbb{S} \times \mathbb{S})) \\
\gamma_{\text{SSA}^\#}(p^\#, \hat{g}) \triangleq \{(\mathcal{H}_{p^\#}(\hat{s}), \mathcal{H}_{p^\#}(\hat{s}')) \mid (\hat{s}, \hat{s}') \in \gamma_{\hat{\mathbb{G}}}(\hat{g})\} \\
\hline
\mathcal{F}_{\text{SSA}^\#}(p^\#, \hat{g}) \triangleq (p'^\#, \text{graph}(p'^\#)) \quad \text{where } p'^\# = \mathcal{F}_{\mathbb{L} \rightarrow \Sigma_\perp^\#}(p^\#)
\end{array}$$

Fig. 7. The SSA abstract domain : definition (top), concretizations (middle), transfer function (bottom)

Once we have the $\mathcal{H}_{p^\#}$ function, the meaning of the SSA abstraction $(p^\#, \hat{g})$ is given as the image of the transition system $(\hat{\mathbb{S}}, \rightsquigarrow_{\hat{g}})$ by $\mathcal{H}_{p^\#}$. This image is a transition system $(\mathbb{S}, \hookrightarrow)$ where \hookrightarrow is defined as follows:

$$\forall \hat{s}, \hat{s}' \in \hat{\mathbb{S}} : \hat{s} \rightsquigarrow_{\hat{g}} \hat{s}' \Leftrightarrow \mathcal{H}_{p^\#}(\hat{s}) \hookrightarrow \mathcal{H}_{p^\#}(\hat{s}'). \quad (1)$$

This definition implies that $\mathcal{H}_{p^\#}$ is a strong homomorphism from $(\hat{\mathbb{S}}, \rightsquigarrow_{\hat{g}})$ to $(\mathbb{S}, \hookrightarrow)$ (indeed, the \mathcal{H} in $\mathcal{H}_{p^\#}$ stands for homomorphism). The goal of our soundness theorem will be to establish that \hookrightarrow overapproximates \rightsquigarrow (and, since the SSA translation is sound but also complete, we will also prove that \hookrightarrow and \rightsquigarrow are equal). Figure 2 illustrates the $\mathcal{H}_{p^\#}$ homomorphism.

Now as we will see, the original transition system $(\mathbb{S}, \rightsquigarrow)$ and the corresponding SSA transition system $(\hat{\mathbb{S}}, \rightsquigarrow_{\hat{g}})$ only match when both transition systems are restricted to the reachable states. This justifies why in Figure 7 we concretize SSA graphs as a transition system between the reachable SSA states $\hat{\mathbb{S}}^*$. Formally, given \hat{g} , we define $\hat{\mathbb{S}}^*$ as $\hat{\mathbb{S}}^* \triangleq \{\hat{s} \in \hat{\mathbb{S}} : \exists \hat{s}_0 \in \hat{\mathbb{S}}_0, \hat{s}_0 \rightsquigarrow_{\hat{g}}^* \hat{s}\}$ where $\rightsquigarrow_{\hat{g}}^*$ represent the reflexive and transitive closure of $\rightsquigarrow_{\hat{g}}$. We define $\rightsquigarrow_{\hat{g}|\hat{\mathbb{S}}^*} \triangleq \{(\hat{s}, \hat{s}') \in \rightsquigarrow_{\hat{g}} \mid \hat{s} \in \hat{\mathbb{S}}^*\}$.

5.4.3 Transfer functions. The transfer function for the SSA abstraction consists in updating $p^\#$ at every step (using the previously defined $\mathcal{F}_{\mathbb{L} \rightarrow \Sigma_\perp^\#}$), and using the graph operator to accordingly update the SSA graph \hat{g} .

This formal analysis can be made more efficient in practice. Note that the computation of the SSA graph at each iteration does not require the SSA graph at the previous iteration. Thus, in analyses where the SSA abstraction is only used to get an SSA translation, and is not used to improve the precision of other abstract domains, it suffices to call graph once at the end. This is what our evaluation of Section 7.1 does.

If the SSA graph is used in a combination with other domains, we can avoid recomputing the whole SSA graph at each iteration and instead compute the SSA graph incrementally along with a chaotic iteration. In Section 6 we present an algorithm that does this (and also ensures completeness of the SSA abstraction during the fixpoint computation).

5.4.4 Soundness of the analysis. Our main soundness theorem states that once the analysis reaches a fixpoint, the SSA abstraction is sound and complete representation of the transition system of the original program.

THEOREM 5.8 (SOUNDNESS AND COMPLETENESS OF THE SSA ABSTRACTION COMPUTATION).

Let $(p^\#, \hat{g})$ be a fixpoint of $\mathcal{F}_{\text{SSA}^\#}$. Then: $\gamma_{\text{SSA}^\#}(p^\#, \hat{g}) = \rightsquigarrow_{|\mathbb{S}^*}$

PROOF SKETCH. First, we show that $\mathcal{H}_{p^\#}$ is a strong homomorphism from $(\hat{\mathbb{S}}, \rightsquigarrow_{\hat{g}})$ to $(\mathbb{S}, \rightsquigarrow)$. The main idea of this proof is that $\mathcal{E}^\#[\![\cdot]\!](\cdot)$ and $\mathcal{I}^\#[\![\cdot]\!](\cdot)$ are exact operations, and the bindings to symbolic variables created by edge operator compensates the loss of precision introduced by \sqcup^ℓ . Note that $\mathcal{H}_{p^\#}$ is still a strong homomorphism if we use the assignment semantics of SSA (i.e. without the unbind operation) for $\rightsquigarrow_{\hat{g}}$, as, following Theorem 5.1, in a transition from ℓ to ℓ' the unbound variables are those that no longer occur $p^\#[\ell']$.

This strong homomorphism allows proving completeness, but not soundness. Indeed, $p^\#$ does not represent all the states, only the reachable states (i.e. in general $\gamma_{\mathbb{L} \rightarrow \Sigma_\perp^\#}(p^\#) \subset \mathbb{S}$). Because of this, $\rightsquigarrow_{\hat{g}}$ cannot represent all the transitions that \rightsquigarrow can. However, it can represent the transitions between the reachable states. Thus, we restrict \rightsquigarrow to $\rightsquigarrow_{|\mathbb{S}^*} \triangleq \{(s, s') \in \rightsquigarrow \mid s \in \mathbb{S}^*\}$. If we want $\mathcal{H}_{p^\#}$ to remain a strong homomorphism, we also need to restrict $\rightsquigarrow_{\hat{g}}$ to $\rightsquigarrow_{\hat{g}|\hat{\mathbb{S}}^*}$. In that case, $\mathcal{H}_{p^\#}$ becomes a *surjective* strong homomorphism (and hence, a bisimulation [Sangiorgi 2009]) which implies that $\gamma_{\mathbb{S}\mathbb{S}\mathbb{A}^\#} = \rightsquigarrow_{|\mathbb{S}^*}$. \square

The full proof is given in Appendix C.

We can make two observations about this theorem. Firstly, a transition system is one of the most precise semantics in the hierarchy of program semantics [Cousot 2002], being equivalent to the maximal trace semantics¹. Thus, the fact that the SSA abstraction is sound and complete when concretized into a transition system implies that it is also sound and complete for standard semantics such as set of traces, set of input/output relations, or set of reachable states.

Secondly, throughout the proof we are only interested in the soundness and completeness when the fixpoint is reached, which is unlike the usual proof structure in static analyses based on abstract interpretation. Indeed, the concretization of successive iterations of $\mathcal{F}_{\mathbb{S}\mathbb{S}\mathbb{A}^\#}$ is not monotonic, and some intermediate iterations include spurious traces (i.e. are not complete) that disappear when the fixpoint is reached (by contrast, the computation of successive iterations of $\mathcal{F}_{\mathbb{L} \rightarrow \Sigma_\perp^\#}$ was monotonic). The next section fixes this completeness issue in intermediate iterations, but note that the computation of the SSA abstraction will still be non-monotonic.

6 INCREMENTALLY COMPLETE CONSTRUCTION OF SSA

This section first explains why the previous construction of SSA abstraction is not incrementally complete, and provides a practical solution to fix the issue.

6.1 Normal construction of SSA abstraction is not incrementally complete

Our computation of SSA using abstract interpretation is both sound (it describes all the behaviors of the original program) and complete (all the behaviors that it describes are present in the original program). However, it is not *continuously* sound and complete, i.e. intermediate computations may produce abstractions that are neither sound nor complete. While the fact that intermediate computations are not sound is expected (as abstract interpretation grows the abstraction until the fixpoint is reached), the incompleteness of intermediate computations is more problematic. This has practical consequences: if other domains rely on the intermediate SSA construction (e.g. for translation into a logic formula given to an SMT solver [Gurfinkel et al. 2015], to compute an exact abstract transformer [Reps et al. 2004], to retrieve the target of indirect jumps when analyzing machine code [Djoudi et al. 2016], etc.), this may lead to imprecisions of the static analysis.

Consider the example of Figure 8. Figure 8a presents a program and the computation of the term abstraction $p^\#$ after 4 iterations of $\mathcal{F}_{\mathbb{L} \rightarrow \Sigma_\perp^\#}$ (the abstract store $p^\#[\ell]$ is given next to each program

¹It is usual to define the set of traces from a transition system, but we can also define a transition relation from the successive states that appear in all the traces.

883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931

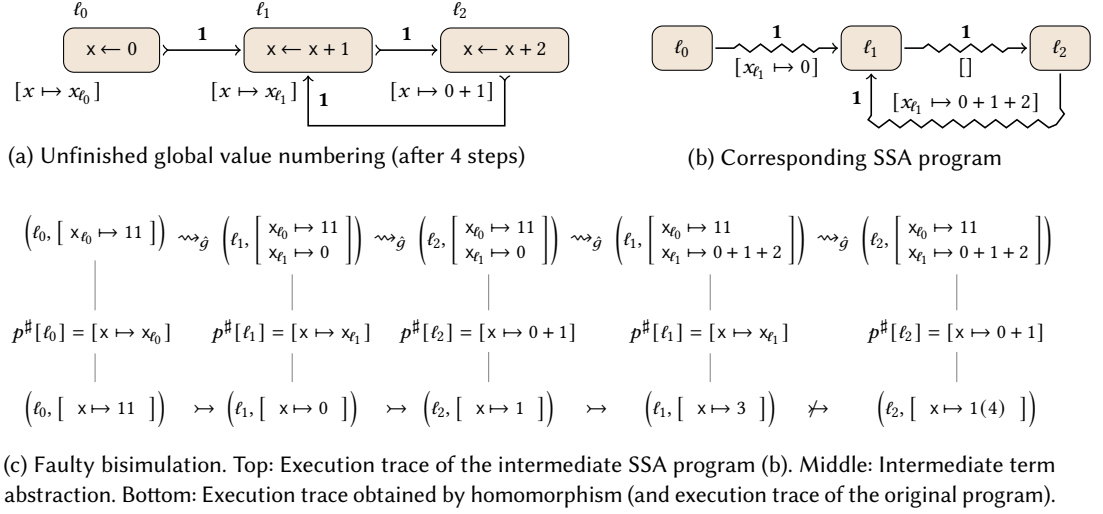


Fig. 8. Example of an incomplete intermediate SSA abstraction, with the wrong execution trace.

location ℓ). We see that the fixpoint computation has joined, at the loop entry ℓ_1 , the abstract stores coming from the beginning of the program ℓ_0 and from the loop back edge ℓ_2 , creating the symbolic variable x_{ℓ_1} , but x_{ℓ_1} has not yet been propagated to ℓ_2 .

Figure 8b presents the corresponding graph $\hat{g} \triangleq \text{graph}(p^\#)$. We can see that the back-edge is labeled with the binding $[x_{\ell_1} \mapsto 0 + 1 + 2]$, which is not the correct binding that we will get at the end of the analysis (the correct binding is $[x_{\ell_1} \mapsto x_{\ell_1} + 1 + 2]$).

However, the fact that the bindings on the SSA graph are "wrong" is only a consequence of the real issue: because the analysis is unfinished, the intermediate versions of $p^\#$ do not represent all the possible program states (e.g., $p^\#[\ell_2]$ only represents the program states at ℓ_2 for the first iteration of the loop); and at the same time, the edges present in \hat{g} allows the transition system that \hat{g} represents to follow paths that lead to program states that are not yet represented by $p^\#$. The erroneous execution trace of Figure 8c illustrates this: the program path $\ell_0 - \ell_1 - \ell_2 - \ell_1 - \ell_2$ leads to a concrete store $[x \mapsto 4]$ which is not represented by $p^\#[\ell_2]$, leading to the fact that this path in the original program cannot be matched in the SSA graph (instead, the path continues with a wrong transition). Note that this problem can also appear in loop-free programs.

6.2 Making SSA construction incrementally complete

To make the SSA construction complete on intermediate steps, we need to limit the SSA graph so that transition to states that are not represented by $p^\#$ is impossible. For instance in Figure 8, if we remove the transition from ℓ_2 to ℓ_1 (for instance, by keeping only the transition between ℓ_0 and ℓ_1), then the SSA program represents a subset of the reachable program transitions of the original program.

Our idea is that the SSA graph should contain an edge from ℓ to ℓ' only if

$$\forall \sigma, \sigma' \in \Sigma : (\ell, \sigma) \in \gamma_{\perp \rightarrow \Sigma_\perp}^\#(p^\#) \wedge (\ell, \sigma) \mapsto (\ell', \sigma') \Rightarrow (\ell', \sigma') \in \gamma_{\perp \rightarrow \Sigma_\perp}^\#(p^\#)$$

i.e. if all states represented by $p^\#[\ell]$, will have the next state σ' be represented by $p^\#[\ell']$. We call such an edge *fully-propagated*. For instance, the edge from ℓ_1 to ℓ_2 in the SSA graph in Figure 8 is

not fully propagated, because $(\ell_1, [x \mapsto 8]) \in \gamma_{\mathbb{L} \rightarrow \Sigma_{\perp}^{\#}}(p^{\#})$ and $(\ell_1, [x \mapsto 8]) \rightsquigarrow (\ell_2, [x \mapsto 9])$, but $(\ell_2, [x \mapsto 9]) \notin \gamma_{\mathbb{L} \rightarrow \Sigma_{\perp}^{\#}}(p^{\#})$. On the contrary, the edge from ℓ_0 to ℓ_1 is fully-propagated.

This idea of a fully-propagated edge (ℓ, ℓ') is that the stores that can be seen at ℓ' include the stores that come from ℓ . In the abstract semantics, it corresponds to an (abstract) inclusion between abstract stores, that we define below.

Definition 6.1 (Order relation on $\Sigma_{\perp}^{\#}$, the abstract stores extended with \perp).

$$\forall \ell \in \mathbb{L}, \forall \sigma_1^{\#}, \sigma_2^{\#}, \sigma_3^{\#} \in \Sigma_{\perp}^{\#} : \quad \perp \sqsubseteq^{\ell} \perp \quad \perp \sqsubseteq^{\ell} \sigma_2^{\#} \quad \sigma_1^{\#} \sqsubseteq^{\ell} \sigma_2^{\#} \triangleq \exists \sigma_3^{\#} : \sigma_2^{\#} = \sigma_1^{\#} \sqcup^{\ell} \sigma_3^{\#}$$

This definition is equivalent to the usual algebraic definition $\sigma_1^{\#} \sqsubseteq^{\ell} \sigma_2^{\#} \triangleq \sigma_2^{\#} = (\sigma_1^{\#} \sqcup^{\ell} \sigma_2^{\#})$ (Lemma D.1), but it is closer to the intended meaning of why the \sqsubseteq^{ℓ} relation is needed in our case (i.e. $\sigma_1^{\#} \sqsubseteq^{\ell} \sigma_2^{\#}$ if we can obtain $\sigma_2^{\#}$ by joining $\sigma_1^{\#}$ with something else).

Equipped with this order relation, we can define the maximal set of fully-propagated edges (or, maximal fully-propagated subgraph) given $p^{\#} \in \mathbb{L} \rightarrow \Sigma_{\perp}^{\#}$ as follows:

$$\max G(p^{\#}) \triangleq \{(\ell, \ell') \in \text{dom } \mathcal{G} \mid \mathcal{G}[\ell, \ell'] = 0 \vee p^{\#}[\ell] = \perp \vee \mathcal{I}^{\#}[\text{instr}[\ell]](p^{\#}[\ell]) \sqsubseteq^{\ell'} p^{\#}[\ell']\}$$

This definition is derived from the definition of $\mathcal{F}_{\mathbb{L} \rightarrow \Sigma_{\perp}^{\#}}$ describing the transfer function over edges in Figure 4. On the example of Figure 8, we have $\max G(p^{\#}) = \{(\ell_0, \ell_1)\}$.

Then, we define a modified transfer function for the SSA abstract domain, consisting in limiting the SSA graph to the maximal fully-propagated subgraph of the current value of $p^{\#}$. Formally:

$$\mathcal{F}'_{\text{SSA}^{\#}}(p^{\#}, \hat{g}) \triangleq \mathcal{F}_{\mathbb{L} \rightarrow \Sigma_{\perp}^{\#}}(p^{\#}), \text{graph}(\mathcal{F}_{\mathbb{L} \rightarrow \Sigma_{\perp}^{\#}}(p^{\#}))|_{\max G(p^{\#})}$$

where we note by $\hat{g}|_G \triangleq \{(\ell, \ell') \mapsto (\hat{e}, B) \mid (\ell, \ell') \mapsto (\hat{e}, B) \in \hat{g} \wedge (\ell, \ell') \in G\}$ the restriction of an SSA graph \hat{g} to the edges in G . By restricting the SSA graph to fully-propagated edges of $p^{\#}$, we can prove that intermediate computation performed by this new transfer function is complete:

THEOREM 6.2 (INCREMENTAL SSA GRAPHS ARE COMPLETE).

Let $p^{\#} \in \mathbb{L} \rightarrow \Sigma_{\perp}^{\#}$, $G = \max G(p^{\#})$, and $\hat{g} = \text{graph}(\mathcal{F}_{\mathbb{L} \rightarrow \Sigma_{\perp}^{\#}}(p^{\#}))$. Then: $\gamma_{\text{SSA}^{\#}}(p^{\#}, \hat{g}|_G) \subseteq \rightsquigarrow_{\mathbb{S}^*}$.

PROOF SKETCH. The proof goes as follows. We show that the notion of *fully-propagated graph* (a set of fully-propagated edges) G corresponds to being a post-fixpoint of the original program restricted to G . For this, we define the notion of post-fixpoint and show that the theorems that we proved when $p^{\#}$ is a fixpoint also apply when $p^{\#}$ is a post-fixpoint, and in particular the soundness and completeness of $p^{\#}, \hat{g}$ when it is a (post)fixpoint of the SSA analysis (Theorem C.10). Using this theorem and the fact that we have a post-fixpoint of the original program restricted to G , we prove that $\gamma_{\text{SSA}^{\#}}(p^{\#}, \hat{g}|_G)$ is transition system of the original program limited to G , which is a subset of the original transition system: thus our SSA abstraction does not have spurious behaviours. \square

The full proof is in Appendix D page 37.

6.3 A practical algorithm for incremental computation of SSA graphs

The fixpoint computation using $\mathcal{F}'_{\text{SSA}^{\#}}$ above has two efficiency problems. The first is that it recomputes the whole $p^{\#}$ and \hat{g} at every step even though these do not change much between two iterations. The solution for computing $p^{\#}$ incrementally is known, and consists in performing chaotic iteration [Bourdoncle 1993; Cousot 1977]. It is quite easy to also compute the incomplete version of the graph incrementally (i.e., make the transfer functions of Figure 7 incremental) by updating all the edges going to ℓ when $p^{\#}[\ell]$ is updated. However, it is more difficult to ensure that \hat{g} is complete, and in particular to ensure that \hat{g} only contains the edges of $\max G(p^{\#})$.


```

981 Input: A weak topological ordering
982       (WTO) of locations  $wto$ 
983 Output: A program abstraction ( $p^\#$ )
984           and SSA graph ( $\hat{g}$ )
985 Main algorithm is
986    $p^\# := []$ ;
987    $\hat{g} := \{\}$ ;
988   sequence( $wto$ );
989
990 Procedure sequence( $list$ ) is
991   forall  $elt \in list$  do
992     if  $elt$  is a single location  $\ell$ 
993       then
994         location( $\ell$ );
995       else
996         component( $elt$ );
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029

```

Procedure location(ℓ) **is**
 $p^\#[\ell] := \mathcal{F}_{\mathbb{L} \rightarrow \Sigma_\perp^\#}(p^\#)[\ell]$;
 $\hat{g} := \hat{g} \cup \{\text{edge}(p^\#, \ell_1, \ell) \mid (\ell_1, \ell) \in \text{dom } \mathcal{G}\}$;
Procedure component($head :: tail$) **is**
local variables $save\hat{g}, curhead$;
 $save\hat{g} := \hat{g}$;
location($head$);
repeat
 $curhead := p^\#[head]$;
 $\hat{g} := save\hat{g} \cup$
 $\{\text{edge}(p^\#, \ell, head) \mid (\ell, head) \in \text{dom } \mathcal{G}$
 $\quad \wedge \ell \leq head\}$;
sequence($tail$);
location($head$);
until $curhead = p^\#[head]$;

Fig. 9. A practical algorithm for incremental computation of an SSA abstraction.

Our solution to this problem (Figure 9) is to rely on a property of the recursive fixpoint iteration strategy of Bourdoncle [1993], which is that when ℓ_c is the currently modified location and $p^\#$ the current program abstraction, the subgraph $\{(\ell, \ell') \in \mathbb{L} \times \mathbb{L} \mid \ell \leq \ell_c \wedge \ell' \leq \ell_c\}$ is a fully-propagated subgraph of $p^\#$ (here, \leq represents the weak topological ordering of Bourdoncle [1993]). Our algorithm builds on this observation and incrementally compute \hat{g} so that the edges of \hat{g} are limited to this subgraph. In more details, the algorithm mostly implements Bourdoncle’s recursive iteration strategy, taking as input a WTO (which is a sequence of locations or (connected) components, where a component is a head location followed by a sequence of locations or components). The main change compared to the normal recursive iteration strategy is that the SSA graph \hat{g} is saved in a snapshot ($save\hat{g}$) before processing a component, and this snapshot is retrieved when another iteration on the component is needed. The feedback edges (edges from later locations going to the head of the component) do not appear in the \hat{g} graph until the fixpoint iteration over the component is over. The end result is that this algorithm continuously produces (e.g. before the call to every location function) a complete SSA abstraction of the program (and moreover, terminates with a sound and complete SSA abstraction of the program).

7 USE CASES

7.1 Single-pass optimized translations to SSA

Implementation. In the first use case, we demonstrate the feasibility of doing SSA translation using abstract interpretation. We have implemented the algorithms described in the paper as a small plugin for the FRAMA-C platform and used the LLVM intermediate representation (which is in SSA form) as the target of our translation. In particular, we have implemented the symbolic expression analysis of Section 4, followed by a call to the graph operator, modified to directly produce LLVM code instead of SSA graphs. An important (but small) change compared to our formalization is that we have replaced guards and non-determinism with terminator instructions (i.e., branches and conditional branches). Following the formal definition of our analysis, the implementation of this plugin does SSA translation, Global value numbering, Global code motion, but also Sparse Conditional

1030 Constant Propagation [Wegman and Zadeck 1991] (by rewriting constant terms when it can), all in
1031 only 970 lines of OCaml code (measured by Sloccount), excluding a 316 lines implementation of
1032 Okasaki's Patricia tries [Okasaki and Gill 1998], and code implementing the iterative and recursive
1033 iteration strategy on Bourdoncle's [Bourdoncle 1993] WTO (183 lines). The parser and WTO
1034 calculation is provided by FRAMA-C, and the LLVM translation by the standard LLVM OCaml
1035 bindings. The goal of this experiment is to demonstrate that doing SSA translation via abstract
1036 interpretation is not only of theoretical interest but that it can be done in practice as the translation
1037 achieves reasonable performances. The full description of this implementation and the results of
1038 the evaluation is provided in Appendix E, while the code for the experiment and its results can be
1039 retrieved at <https://doi.org/10.5281/zenodo.7152678>. Here we reproduce only the main elements
1040 and our conclusions.

1041
1042
1043 *Experimental settings.* We have used this implementation to evaluate both the soundness and
1044 performance of our algorithm. To avoid the problem of addressable variables in C that cannot be
1045 directly translated to SSA, we have used the CSMITH [Yang et al. 2011] C generator, using options to
1046 generate only non-addressable integer variables and complex unstructured control flow, to generate
1047 100 large C files. We used our plugin to convert the C code to LLVM code, then used LLVM (version
1048 13) to compile it to native x86 code, that we executed.

1049
1050
1051 *Results and conclusions.* First, all the C code without undefined behavior that we generated,
1052 returned the same value as the one compiled using a standard compiler. This confirms that SSA
1053 translation using abstract interpretation can be used to soundly translate programs to SSA, which is
1054 not a surprise since we provided the proof for all of our algorithms.

1055 Furthermore, *the time spent doing so is in the order of magnitude compatible for incorporation in a*
1056 *compiler:* if we add incorporated our SSA translation pass inside clang, the maximum observed
1057 slowdown compared to a native code compilation by clang would have been $\times 6.57$, which is slower
1058 but acceptable (the median slowdown would have been $\times 4.30$). This is a maximum, as such an
1059 incorporation would lead to removing existing passes in clang; furthermore, additional engineering
1060 (such as switching from OCaml to C) could further improve the performance. *One of the reason for*
1061 *this efficiency is that fixpoint iteration with the symbolic expression domain is usually obtained using*
1062 *few iterations*, even in large programs: in the iterative fixpoint iteration strategy of Bourdoncle
1063 [Bourdoncle 1993], we never needed more than 6 iterations to reach the fixpoint on any strongly
1064 connected component. The other part is explained by the use of functional maps for low-complexity
1065 operations on abstract stores. *One crucial engineering decision in our analysis performance was the*
1066 *use of Okasaki maps [Okasaki and Gill 1998] to implement the abstract stores*, instead of the more
1067 common binary search trees (as used in Astrée [Blanchet et al. 2002]), as otherwise the time spent
1068 in the \sqcup^l operation dominates the time spent in the analysis. Okasaki maps allow more efficient
1069 handling of the parts of the abstract store that have not changed between when the different states
1070 that are joined with \sqcup^l . Finally, *combining SSA translation with other analyses such as constant*
1071 *propagation and dead code elimination can significantly improve the time spent in the analysis:* in this
1072 case, incorporating our pass in clang would make it only 4.73 time slower (instead of 6.57 times),
1073 at worst (and the median slowdown would have been $\times 3.19$)

1074 *Whether, with careful engineering, implementing compilers as a single-pass combination of modular*
1075 *abstract domains, can outperform the existing designs as a sequence of passes, is an open question*, but
1076 our initial results indicate that the single-pass design could be competitive in terms of speed of
1077 compilation.

1078

7.2 A static analyzer for C and machine code

The main motivation for this work is to describe the working of CODEX, a multi-language static analyzer that can process both C (as a FRAMA-C plugin) and machine code (as a BINSEC plugin) that was used as the basis of the experiments in Nicole et al. [2021b,a, 2022]. CODEX analyses an input program by combining the symbolic expression and SSA abstract domains with control-flow, numeric and memory abstractions. The symbolic expressions and SSA abstract domains have three main uses:

- (1) The symbolic expressions allow proving equality between values. Combined with term rewriting rules, it can be used to prove the preservation of complex predicates; one application is proving preservation of some OS kernel invariants in Nicole et al. [2021b].
- (2) It is used to perform a sparse [Tavares et al. 2014; Wegman and Zadeck 1991] numerical analysis of the program simultaneously with its SSA translation. This numerical analysis, in turn, is used to improve the precision of the symbolic expressions, control-flow, and memory abstractions. The benefit of such a sparse analysis is that it is more efficient in terms of memory and CPU time, and that it allows propagating constraints across different statements, which is important when analyzing machine code (that consists of a lot of small instructions).
- (3) The SSA translation allows generating SMT or Horn queries. Thanks to the combination of SSA and memory abstractions, this logic query is automatically simplified, which is important as SMT solvers do not handle memory very well [Farinier et al. 2018]. Thanks to the incrementally complete construction of SSA (Section 6), the SMT queries can be performed anytime during the analysis, and is used in particular to refine Boolean expressions such as assertions or opaque predicates, or to enumerate control-flow targets when needed.

The full description of this analysis and combination of domains is out of the scope of this paper, but the above description motivates the practical benefits of using the symbolic expressions and SSA abstractions.

8 RELATED WORK

8.1 Abstract interpretation on cyclic terms and computing the global value graph

Having terms representing expressions or predicates inside an abstract domain is a natural idea already present in the seminal paper of Kildall [1973]. The main difficulty of using terms in a static analysis is dealing with cycles of the term (coming from loops in the original program), and ensuring termination. Indeed, many analyses (e.g., symbolic execution [King 1976], bounded model checking [Clarke et al. 2004], weakest precondition and strongest-postcondition [de Bakker and Meertens 1975; Dijkstra 1975; Leino 2005] calculus) transform the program into a single global term or formula, but sidestep the problem of termination by analyzing only loop-free subsets of the program (although they can be viewed as abstract interpretations [Cousot 1978, ch.3.3]).

Some static analyses [Gulwani and Necula 2004; Kildall 1973] include terms that refer to program variables instead of symbolic variables. This generally removes the need for the terms to be cyclic. However, cyclicity reappears when program variables are substituted with their definition; Miné [2006] solves this problem by using a special \top variable where we would use symbolic variables, thereby losing some precision. Some static analyses [Chang and Leino 2005; Chang and Rival 2013; Gange et al. 2016; Illous et al. 2021] create fresh variables when needed, which is precise but requires special care to ensure termination when the program has loops. Their solution is to use renaming: in particular, inclusion to test if a post-fixpoint is reached is done modulo renaming of the variables in the abstract elements. One issue with renaming is that it makes the definition and implementation of an abstract domain more complex. Furthermore, these solutions can be used

1128 to get acyclic terms at different program points, but not to create one global cyclic term like the
 1129 global value graph.

1130 The most common cyclic term graph used by static analyses is the global value graph. The classic
 1131 solution to get this graph is to do an SSA translation, which solves the problem of termination of
 1132 the analysis because SSA translation is done as a program translation [Cytron et al. 1991] rather
 1133 than as an abstract interpretation. Analyses using this graph then only need to manipulate SSA
 1134 variables instead of nodes in a cyclic term graph, and, for instance, the problem of global value
 1135 numbering is reduced to computing equivalence classes of equality between SSA variables [Alpern
 1136 et al. 1988; Barthe et al. 2014; Rosen et al. 1988; Rüthing et al. 1999]. The drawback of this design is
 1137 that SSA translation must be done sequentially with GVN and other analyses, instead of having
 1138 mutually beneficial combination of domains.

1139 Some papers predating SSA [Reif and Lewis 1986; Reif and Tarjan 1982] did build the global value
 1140 graph and performed global value numbering directly on the original program. In particular, [Reif
 1141 and Lewis 1986] provided a fixpoint characterization of the global value graph, and introduced
 1142 mappings from program expressions to symbolic expressions (playing the same role as our symbolic
 1143 expression abstraction $p^\#$). However, they dismissed the computation of the global value graph
 1144 using dataflow analysis (proposing a complex sparse algorithm instead) for fear of it being too slow,
 1145 and because their characterization did not lead to a sufficiently precise value graph; we solve these
 1146 issues using efficient abstract interpretation techniques [Blanchet et al. 2002; Bourdoncle 1993;
 1147 Cousot 1977], and by combining creation of the global value graph with global value numbering.

1148 *We propose to solve the problem of having cyclic terms in an abstract domain by introducing*
 1149 *recursion variables [Ariola and Klop 1996] that have a deterministic name, and have this name follow*
 1150 *the name of the recursion variables that already exists in the original program, i.e. the names of*
 1151 *the control-flow locations. We apply this method to build the global value graph using a standard*
 1152 *flow-sensitive analysis on the original program, where the graph is simplified by a simultaneous global*
 1153 *value numbering. The method does not require a prior SSA translation (instead, it is used to perform*
 1154 *SSA translation).*

1155
 1156

1157 8.2 SSA translation and reconstruction

1158 Since the invention of SSA form [Rosen et al. 1988], several algorithms have been created to compute
 1159 SSA form. The classic algorithm of Cytron et al. [1991] works in two phases, first computing where
 1160 to insert ϕ functions after computation of the dominance frontier, then applying a linear pass that
 1161 assigns unique names to the different definitions of each variable. Sreedhar and Gao [1995] makes
 1162 the ϕ -function insertion pass linear by proposing a data structure called DJ-graph, that builds upon
 1163 the dominator tree. The algorithm of Aycock and Horspool [2000] performs a pessimistic insertion
 1164 of ϕ -functions and then improves the translation using rewriting rules. The ϕ -function insertion
 1165 phase of these algorithms is related to our symbolic expression analysis, as the introduction of
 1166 symbolic variables introduced during a \sqcup^ℓ operation corresponds to the insertion of a ϕ node at ℓ .

1167 Some algorithms can work directly from programs in AST form [Brandis and Mössenböck 1994;
 1168 Braun et al. 2013], without requiring a prior analysis, like computation of the domination tree. In
 1169 particular, the algorithm of Braun et al. [2013] produces minimal SSA on unstructured programs,
 1170 and can perform "on the fly optimizations" (which are similar to rewriting terms in our symbolic
 1171 expression analysis) to improve the precision and efficiency of SSA generation.

1172 *SSA translation based on abstract interpretation improves on the latter algorithm by performing "on*
 1173 *the fly" global value numbering, sparse conditional constant propagation, or any optimization based*
 1174 *on semantic properties obtained by combination with an abstract domain, in addition to not requiring*
 1175

1176

1177 *a priori analysis. It is also the first algorithm to do SSA translation incrementally while maintaining*
1178 *completeness.*

1179 “Maintaining SSA is often one of the more complicated and error-prone part in [program]
1180 optimizations” [Hack 2016]. Combing our SSA abstract domain with another domain, and viewing a
1181 program transformation as a reduced product [Cousot and Cousot 1979] between the two domains,
1182 is an interesting solution to this problem that can be used to create transformation passes that
1183 automatically maintain SSA.

1184

1185

8.3 Syntax and semantics of SSA programs

1186 *We represent a program in SSA form as a graph whose nodes are control locations and edges contain*
1187 *guards and bindings to symbolic variables, where the bindings correspond to ϕ functions in traditional*
1188 *SSA. We have proposed a semantics where SSA states consists of valuations of the symbolic variables*
1189 *that are in scope, so that SSA states and original program states are made bisimilar using the computed*
1190 *symbolic expressions as a bisimulation (and homomorphism) relation.*

1191 Usually, SSA is viewed as reusing the original program syntax, and SSA transformation consists
1192 in renaming the program variables and inserting the necessary ϕ functions. The SSA semantics
1193 is based on the assignment of SSA variables, and ϕ functions are also seen as doing assignments
1194 [Barthe et al. 2014; Cytron et al. 1991; Mansky and Gunter 2010; Zhao et al. 2012].

1195 The idea of binding (instead of assigning) SSA variables to unbind those that go out of scope also
1196 exists when SSA is viewed as a functional program [Appel 1998b; Kelsey 1995], where the scope
1197 is represented explicitly in the syntax (rather than implicitly by domination). Schneider [2013]
1198 showed that both interpretations (one based on assignments and the other on bindings) can be
1199 used on functional representations of SSA. *We showed that we can also have both interpretations on*
1200 *a graph-based language without an explicit syntactic scope.* We preferred using a graph as the syntax
1201 for SSA rather than lambda terms, as graphs can be built incrementally using abstract domain
1202 operations, which seems more difficult to do when the abstraction is a lambda term.

1203 The idea of not attaching sub-terms to a specific control location was proposed by Click and
1204 Paleczny [1995] in an SSA representation called Sea of Nodes, which is the closest to our SSA
1205 graphs (in particular, to a variant of our graphs that uses terminator instructions instead of guards
1206 + nondeterminism). The Sea of Nodes can be described as a low-level description of our SSA graphs,
1207 where their nodes would correspond to the memory blocks used to represent the graph and the
1208 edges to the pointers. Demange et al. [2018] proposed a formal semantics for Sea of Nodes using a
1209 mixture of big-step semantics for expressions and small-step semantics for instructions which is
1210 similar to our semantics: the main changes are that we propose to unbind variables that go out of
1211 scope, and we take advantage of the simpler, higher-level syntax of SSA graphs.

1212 An important characteristic of SSA syntax and semantics, and of the SSA translation algorithm, is
1213 simplicity, particularly when the goal is to formally verify SSA transformations [Barthe et al. 2014;
1214 Buchwald et al. 2016; Zhao et al. 2012]. The entire syntax and semantics of our SSA abstraction fit in
1215 Figure 5, and our SSA translation algorithm is completely and formally described in a dozen of lines
1216 (coming from parts of Figure 4 and Figure 6), and are thus arguably very simple. Furthermore, this
1217 description includes GVN and dead code elimination, and other optimizations can be described in a
1218 very compact way as a reduced product with other abstract domains. It would thus be interesting
1219 to investigate if SSA translation based on abstract interpretation can be an effective basis for formal
1220 verification of SSA transformations.

1221

1222

8.4 SSA and static analysis

1223 *One of the motivations for our work is to combine the SSA abstraction with other abstractions. Such a*
1224 *combination benefit from interplay between SSA and static analyses that already exist.*

1225

1226 One of the benefits of SSA form is that it allows more efficient *sparse* static analyses [Choi
1227 et al. 1991] (see, e.g., Tavares et al. [2014] for partitioned per variable analyses, and Mirliaz and
1228 Pichardie [2022] for analyses using relational domains). *Combining our SSA domain with a sparse
1229 analysis allows performing sparse analyses simultaneously with the SSA translation, and integrating
1230 sparse analyses in a standard [Blanchet et al. 2002; Blazy et al. 2017; Journault et al. 2019] abstract
1231 interpretation framework.*

1232 The need for static analyses and, in particular, memory, pointer or alias analyses to improve
1233 the precision of SSA is well-known. This requirement is very important, for instance, when the
1234 SSA form is used to generate logic formulas to an SMT solver [Clarke et al. 2004; Gurfinkel et al.
1235 2015] to increase precision and decrease solving time. Static analyses are even indispensable for
1236 programs where the control-flow is not known in advance because usual SSA translations require a
1237 CFG (e.g., to compute dominance frontiers). In particular, this is the case when analyzing machine
1238 code, i.e. when doing decompilation [Brumley et al. 2013] to SSA, which is a popular target for
1239 decompilation [Van Emmerik 2007; Yadavalli and Smith 2019]. Precisely recovering control-flow in
1240 machine code requires doing control-flow and value analysis simultaneously [Bardin et al. 2011;
1241 Kinder et al. 2009]; and SSA is a good intermediate representation to do this value analysis. To
1242 deal with this phase ordering problem, Van Emmerik [2007] used a simple but inefficient method
1243 consisting in redoing the whole SSA decompilation every time an indirect jump target is discovered
1244 in a function. *Doing SSA translation as an abstract interpretation solves these phase ordering problems:
1245 we can use the incrementally built (Section 6) SSA representation to help the memory, numeric, or
1246 control-flow analyses that improve or are necessary to build the SSA representation.*

1247 Outside SSA, is it well-known that symbolic domains can be used to improve the precision
1248 of abstract domains [Gange et al. 2016; Miné 2006], and our symbolic expression domain can be
1249 used for the same purpose (e.g., to detect equalities between variables that would be missed by
1250 numerical domains). Finally, Cousot and Cousot [2002] explains how abstract interpretation can
1251 help design and prove correct program transformations, and discusses the possibility of doing some
1252 transformations, like constant propagation and dead code elimination, using dataflow analysis.
1253

1254 9 CONCLUSION

1255 SSA translation is not only a program transformation; it is the result of a static analysis based on
1256 abstract interpretation. This analysis has two parts: symbolic expression analysis, which maps
1257 program variables to symbolic expressions; and SSA graph creation, which expresses how the
1258 symbolic variables in the symbolic expressions change according to the control flow. The symbolic
1259 expression analysis (that also does a global value numbering) relates the SSA program graph to the
1260 original program graph by providing a strong homomorphism, a relation that implies bisimulation.
1261 Due to this bisimulation, the SSA-translation domain computes an abstraction of the original
1262 program which is both sound and complete. Furthermore, we can modify the analysis so that
1263 all intermediate abstractions are also complete. This static analysis can be done using relatively
1264 standard dataflow frameworks, which allows combination [Cousot and Cousot 1979] with other
1265 abstract domains. We explained, in this paper, how numerical abstract domains can strengthen
1266 the symbolic expression analysis and SSA abstractions, notably by detecting dead transitions or
1267 more equalities between symbolic expressions. Using a reduced product, we can build optimizing
1268 transformations from a source program to SSA in a single pass.
1269

1270 We have only sketched how the SSA abstraction can strengthen numerical abstract domains,
1271 and did not mention how the SSA abstraction can be combined with memory abstractions to do
1272 register promotion; these studies should be the target of future publications.
1273

REFERENCES

- 1275
1276 Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. 1988. Detecting Equality of Variables in Programs. In *15th ACM*
1277 *Symposium on Principles of Programming Languages (POPL 1988)*. 1–11. <https://doi.org/10.1145/73560.73561>
- 1278 Andrew W Appel. 1998a. *Modern compiler implementation in ML*. Cambridge University Press. [https://doi.org/10.1017/](https://doi.org/10.1017/CBO9780511811449)
1279 [CBO9780511811449](https://doi.org/10.1017/CBO9780511811449)
- 1280 Andrew W. Appel. 1998b. SSA is Functional Programming. *SIGPLAN Notices* 33, 4 (apr 1998), 17–20. [https://doi.org/10.](https://doi.org/10.1145/278283.278285)
1281 [1145/278283.278285](https://doi.org/10.1145/278283.278285)
- 1282 Zena M Ariola and Jan Willem Klop. 1996. Equational term graph rewriting. *Fundamenta Informaticae* 26, 3, 4 (1996),
1283 207–240. <https://doi.org/10.3233/FI-1996-263401>
- 1284 John Aycock and R. Nigel Horspool. 2000. Simple Generation of Static Single-Assignment Form. In *9th International Conference*
1285 *on Compiler Construction – CC 2000 (LNCS, Vol. 1781)*. Springer, 110–124. https://doi.org/10.1007/3-540-46423-9_8
- 1286 Sébastien Bardin, Philippe Herrmann, and Franck Védrine. 2011. Refinement-Based CFG Reconstruction from Unstructured
1287 Programs. In *12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2011,*
1288 *Vol. 6538)*. Springer, 54–69. https://doi.org/10.1007/978-3-642-18275-4_6
- 1289 Gilles Barthe, Delphine Demange, and David Pichardie. 2014. Formal Verification of an SSA-Based Middle-End for CompCert.
1290 *ACM Trans. Program. Lang. Syst.* 36, 1 (2014), 4:1–4:35. <https://doi.org/10.1145/2579080>
- 1291 B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. 2002. Design and implementation
1292 of a special-purpose static program analyzer for safety-critical real-time embedded software. *The Essence of Computation:*
1293 *Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones* 2566 (Oct. 2002), 85–108. [https://doi.org/10.1007/3-](https://doi.org/10.1007/3-540-36377-7_5)
1294 [540-36377-7_5](https://doi.org/10.1007/3-540-36377-7_5)
- 1295 Sandrine Blazy, David Bühler, and Boris Yakobowski. 2017. Structuring Abstract Interpreters Through State and Value
1296 Abstractions. In *18th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2017,*
1297 *Vol. 10145)*, Ahmed Bouajjani and David Monniaux (Eds.). Springer, 112–130. https://doi.org/10.1007/978-3-319-52234-0_7
- 1298 François Bourdoncle. 1993. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and*
1299 *their Applications*. Springer, 128–141. <https://doi.org/10.1007/BFb0039704>
- 1300 Martin Brain, Saurabh Joshi, Daniel Kroening, and Peter Schrammel. 2015. Safety Verification and Refutation by k-
1301 Invariants and k-Induction. In *Static Analysis (SAS 2015)*, Sandrine Blazy and Thomas Jensen (Eds.). Springer, 145–161.
1302 https://doi.org/10.1007/978-3-662-48288-9_9
- 1303 Marc M. Brandis and Hanspeter Mössenböck. 1994. Single-Pass Generation of Static Single-Assignment Form for Structured
1304 Languages. *ACM Trans. Program. Lang. Syst.* 16, 6 (1994), 1684–1698. <https://doi.org/10.1145/197320.197331>
- 1305 Matthias Braun, Sebastian Buchwald, Sebastian Hack, Roland Leißa, Christoph Mallon, and Andreas Zwinkau. 2013. Simple
1306 and Efficient Construction of Static Single Assignment Form. In *22nd International Conference on Compiler Construction*
1307 *(CC 2013)*. https://doi.org/10.1007/978-3-642-37051-9_6
- 1308 David Brumley, JongHyup Lee, Edward J. Schwartz, and Maverick Woo. 2013. Native x86 Decompilation Using Semantics-
1309 Preserving Structural Analysis and Iterative Control-Flow Structuring. In *22th USENIX Security Symposium*. USENIX
1310 Association, 353–368. <https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/schwartz>
- 1311 Sebastian Buchwald, Denis Lohner, and Sebastian Ullrich. 2016. Verified construction of static single assignment form. In
1312 *25th International Conference on Compiler Construction (CC 2016)*. ACM, 67–76. <https://doi.org/10.1145/2892208.2892211>
- 1313 David Bühler, Pascal Cuoq, Boris Yakobowski, Matthieu Lemerre, André Maroneze, Valentin Perrelle, and Virgile Prevosto.
1314 2017. *The EVA plug-in*. Technical Report. CEA LIST, Software Reliability Laboratory, Saclay, France, F-91191. [https:](https://frama-c.com/download/frama-c-eva-manual.pdf)
1315 [//frama-c.com/download/frama-c-eva-manual.pdf](https://frama-c.com/download/frama-c-eva-manual.pdf)
- 1316 Bor-Yuh Evan Chang and K. Rustan M. Leino. 2005. *Abstract Interpretation with Alien Expressions and Heap Structures*.
1317 Springer, 147–163. https://doi.org/10.1007/978-3-540-30579-8_11
- 1318 Bor-Yuh Evan Chang and Xavier Rival. 2013. Modular Construction of Shape-Numeric Analyzers. In *Festschrift for*
1319 *Dave Schmidt (EPTCS, Vol. 129)*, Anindya Banerjee, Olivier Danvy, Kyung-Goo Doh, and John Hatcliff (Eds.). [https:](https://doi.org/10.48550/arXiv.1309.5138)
1320 [//doi.org/10.48550/arXiv.1309.5138](https://doi.org/10.48550/arXiv.1309.5138)
- 1321 Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. 1991. Automatic Construction of Sparse Data Flow Evaluation Graphs.
1322 In *18th ACM Symposium on Principles of Programming Languages (POPL 1991)*, David S. Wise (Ed.). 55–66. [https:](https://doi.org/10.1145/99583.99594)
1323 [//doi.org/10.1145/99583.99594](https://doi.org/10.1145/99583.99594)
- 1324 Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *10th International*
1325 *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, Kurt Jensen and Andreas
1326 Podolski (Eds.). Springer, 168–176. https://doi.org/10.1007/978-3-540-24730-2_15
- 1327 Cliff Click. 1995. Global Code Motion / Global Value Numbering. In *Conference on Programming Language Design and*
1328 *Implementation (PLDI 1995)*, David W. Wall (Ed.). ACM, 246–257. <https://doi.org/10.1145/207110.207154>
- 1329 Cliff Click and Keith D. Cooper. 1995. Combining Analyses, Combining Optimizations. *ACM Trans. Program. Lang. Syst.* 17,
1330 2 (1995), 181–196. <https://doi.org/10.1145/201059.201061>

- 1324 Cliff Click and Michael Paleczny. 1995. A simple graph-based intermediate representation. *ACM Sigplan Notices* 30, 3 (1995),
 1325 35–49. <https://doi.org/10.1145/202530.202534>
- 1326 Patrick Cousot. 1977. *Asynchronous iterative methods for solving a fixed point system of monotone equations in a complete*
 1327 *lattice*. Technical Report. Laboratoire IMAG, Université scientifique et médicale de Grenoble, Grenoble, France. <https://www.di.ens.fr/~cousot/publications.www/Cousot-IMAG-RR88-Sep-1977.pdf>
- 1328 Patrick Cousot. 1978. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un*
 1329 *treillis, analyse sémantique de programmes (in French)*. Ph.D. Dissertation. Université Joseph Fourier, Grenoble, France.
 1330 <https://www.di.ens.fr/~cousot/publications.www/CousotTheseEsSciences1978.pdf>
- 1331 Patrick Cousot. 2002. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation.
 1332 *Theor. Comput. Sci.* 277, 1-2 (2002), 47–103. [https://doi.org/10.1016/S0304-3975\(00\)00313-3](https://doi.org/10.1016/S0304-3975(00)00313-3)
- 1333 Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by
 1334 construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages (POPL 1977)*,
 238–252. <https://doi.org/10.1145/512950.512973>
- 1335 Patrick Cousot and Radhia Cousot. 1979. Systematic design of program analysis frameworks. In *6th ACM Symposium on*
 1336 *Principles of Programming Languages (POPL 1979)*, 269–282. <https://doi.org/10.1145/567752.567778>
- 1337 Patrick Cousot and Radhia Cousot. 2002. Systematic design of program transformation frameworks by abstract interpretation.
 1338 In *29th Symposium on Principles of Programming Languages (POPL 2002)*, John Launchbury and John C. Mitchell (Eds.).
 ACM, 178–190. <https://doi.org/10.1145/503272.503290>
- 1339 Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2006.
 1340 Combination of Abstractions in the ASTRÉE Static Analyzer. In *Revised Selected Papers from the 11th Asian Computing*
 1341 *Science Conference on Advances in Computer Science - Secure Software and Related Issues – ASIAN 2006 (Lecture Notes in*
 1342 *Computer Science, Vol. 4435)*. Springer, 272–300. https://doi.org/10.1007/978-3-540-77505-8_23
- 1343 Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static
 1344 Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (oct 1991), 451–490.
<https://doi.org/10.1145/115372.115320>
- 1345 Jacobus Willem de Bakker and Lambert Meertens. 1975. On the Completeness of the Inductive Assertion Method. *J. Comput.*
 1346 *Syst. Sci.* 11, 3 (1975), 323–357. [https://doi.org/10.1016/S0022-0000\(75\)80056-0](https://doi.org/10.1016/S0022-0000(75)80056-0)
- 1347 Delphine Demange, Yon Fernández de Retana, and David Pichardie. 2018. Semantic reasoning about the sea of nodes. In
 1348 *27th International Conference on Compiler Construction (CC 2018)*, Christophe Dubach and Jingling Xue (Eds.). ACM,
 163–173. <https://doi.org/10.1145/3178372.3179503>
- 1349 Edsger W. Dijkstra. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM* 18, 8
 1350 (aug 1975), 453–457. <https://doi.org/10.1145/360933.360975>
- 1351 Adel Djoudi, Sébastien Bardin, and Éric Goubault. 2016. Recovering High-Level Conditions from Binary Programs. In *21st*
 1352 *International Symposium on Formal Methods (FM 2016)*, 235–253. https://doi.org/10.1007/978-3-319-48989-6_15
- 1353 Benjamin Farinier, Robin David, Sébastien Bardin, and Matthieu Lemerre. 2018. Arrays Made Simpler: An Efficient, Scalable
 1354 and Thorough Preprocessing. In *22nd International Conference on Logic for Programming, Artificial Intelligence and*
 1355 *Reasoning (LPAR-22, Vol. 57)*, Gilles Barthe, Geoff Sutcliffe, and Margus Veanes (Eds.). EasyChair, 363–380. <https://doi.org/10.29007/dc9b>
- 1356 Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. 2016. An Abstract Domain of
 1357 Uninterpreted Functions. In *Verification, Model Checking, and Abstract Interpretation (VMCAI 2016)*, Barbara Jobstmann
 1358 and K. Rustan M. Leino (Eds.). Springer, 85–103. https://doi.org/10.1007/978-3-662-49122-5_4
- 1359 Sumit Gulwani and George C. Necula. 2004. A Polynomial-Time Algorithm for Global Value Numbering. In *Static Analysis*
 1360 *Symposium (SAS 2004)*, Roberto Giacobazzi (Ed.). Springer, 212–227. https://doi.org/10.1007/978-3-540-27864-1_17
- 1361 Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A Navas. 2015. The SeaHorn verification framework. In
 1362 *International Conference on Computer Aided Verification (CAV 2015)*. Springer, 343–361. https://doi.org/10.1007/978-3-319-21690-4_20
- 1363 Sebastian Hack. 2016. *SSA reconstruction* (1st ed.), Chapter 5. In [Rastello 2016].
- 1364 Julien Henry, David Monniaux, and Matthieu Moy. 2012. PAGAL: a path sensitive static analyzer. In *Tools for Automatic*
 1365 *Program Analysis (TAPAS)*, Bertrand Jeannet (Ed.). <https://doi.org/10.1016/j.entcs.2012.11.003>
- 1366 Hugo Illous, Matthieu Lemerre, and Xavier Rival. 2021. A relational shape abstract domain. *Formal Methods Syst. Des.* 57, 3
 1367 (2021), 343–400. <https://doi.org/10.1007/s10703-021-00366-4>
- 1368 Matthieu Journault, Antoine Miné, Raphaël Monat, and Abdelraouf Ouadjaout. 2019. Combinations of Reusable Abstract
 1369 Domains for a Multilingual Static Analyzer. In *11th International Conference on Verified Software Theories, Tools, and*
 1370 *Experiments - Revised Selected Papers (VSTTE 2019, Vol. 12031)*, Supratik Chakraborty and Jorge A. Navas (Eds.). Springer,
 1–18. https://doi.org/10.1007/978-3-030-41600-3_1
- 1370 Richard Kelsey. 1995. A Correspondence between Continuation Passing Style and Static Single Assignment Form. In
 1371 *Proceedings ACM SIGPLAN Workshop on Intermediate Representations (IR'95)*, Michael D. Ernst (Ed.). ACM, 13–23.

- 1373 <https://doi.org/10.1145/202529.202532>
- 1374 Gary A Kildall. 1973. A unified approach to global program optimization. In *1st annual ACM SIGACT-SIGPLAN Symposium on Principles of programming languages (POPL 1973)*. <https://doi.org/10.1145/512927.512945>
- 1375 Johannes Kinder, Florian Zuleger, and Helmut Veith. 2009. An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries. In *Verification, Model Checking, and Abstract Interpretation (VMCAI 2009)*, Neil D. Jones and Markus Müller-Olm (Eds.). Springer, 214–228. https://doi.org/10.1007/978-3-540-93900-9_19
- 1376 James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394. <https://doi.org/10.1145/360248.360252>
- 1377 K.R.M. Leino. 2005. Efficient weakest preconditions. *Inform. Process. Lett.* 93, 6 (2005), 281–288. <https://doi.org/10.1016/j.ipl.2004.10.015>
- 1380 William Mansky and Elsa L. Gunter. 2010. A Framework for Formal Verification of Compiler Optimizations. In *1st International Conference on Interactive Theorem Proving – ITP 2010 (Lecture Notes in Computer Science, Vol. 6172)*, Matt Kaufmann and Lawrence C. Paulson (Eds.). Springer, 371–386. https://doi.org/10.1007/978-3-642-14052-5_26
- 1381 Antoine Miné. 2006. Symbolic methods to enhance the precision of numerical abstract domains. In *International Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI 2006)*. Springer, 348–363. https://doi.org/10.1007/11609773_23
- 1382 Solène Mirliaz and David Pichardie. 2022. A Flow-Insensitive-Complete Program Representation. In *23th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2022)*. https://doi.org/10.1007/978-3-030-94583-1_10
- 1383 Greg Nelson. 1980. *Techniques for program verification*. Ph.D. Dissertation. Stanford University, CA, USA.
- 1384 Olivier Nicole, Matthieu Lemerre, Sébastien Bardin, and Xavier Rival. 2021b. No Crash, No Exploit: Automated Verification of Embedded Kernels. In *27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2021)*. 27–39. <https://doi.org/10.1109/RTASS2030.2021.00011>
- 1385 Olivier Nicole, Matthieu Lemerre, and Xavier Rival. 2021a. *Binsec/Codex, an abstract interpreter to verify safety and security properties of systems code*. Technical Report. <https://binsec.github.io/assets/publications/papers/2021-rtas-technical-report-analysis.pdf>
- 1386 Olivier Nicole, Matthieu Lemerre, and Xavier Rival. 2022. Lightweight Shape Analysis Based on Physical Types. In *23rd International Conference on Verification, Model Checking, and Abstract Interpretation – VMCAI 2022 (Lecture Notes in Computer Science, Vol. 13182)*, Bernd Finkbeiner and Thomas Wies (Eds.). Springer, 219–241. https://doi.org/10.1007/978-3-030-94583-1_11
- 1387 Chris Okasaki and Andrew Gill. 1998. Fast Mergeable Integer Maps. In *Workshop on ML*. 77–86.
- 1388 Fabrice Rastello. 2016. *SSA-based Compiler Design* (1st ed.). Springer Publishing Company, Incorporated.
- 1389 John H. Reif and Harry R. Lewis. 1986. Efficient Symbolic Analysis of Programs. *J. Comput. Syst. Sci.* 32, 3 (1986), 280–314. [https://doi.org/10.1016/0022-0000\(86\)90031-0](https://doi.org/10.1016/0022-0000(86)90031-0)
- 1390 John H. Reif and Robert Endre Tarjan. 1982. Symbolic Program Analysis in Almost-Linear Time. *SIAM J. Comput.* 11, 1 (1982), 81–93. <https://doi.org/10.1137/0211007>
- 1391 Thomas Reinbacher and Jörg Brauer. 2011. Precise control flow reconstruction using boolean logic. In *11th International Conference on Embedded Software (EMSOFT 2011)*, Samarjit Chakraborty, Ahmed Jerraya, Sanjoy K. Baruah, and Sebastian Fischmeister (Eds.). ACM, 117–126. <https://doi.org/10.1145/2038642.2038662>
- 1392 Thomas W. Reps, Shmuel Sagiv, and Greta Yorsh. 2004. Symbolic Implementation of the Best Transformer. In *5th International Conference on Verification, Model Checking, and Abstract Interpretation – VMCAI 2004 (Lecture Notes in Computer Science, Vol. 2937)*, Bernhard Steffen and Giorgio Levi (Eds.). Springer, 252–266. https://doi.org/10.1007/978-3-540-24622-0_21
- 1393 Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1988. Global value numbers and redundant computations. In *15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 1988)*. 12–27. <https://doi.org/10.1145/73560.73562>
- 1394 Oliver Rüthing, Jens Knoop, and Bernhard Steffen. 1999. Detecting Equalities of Variables: Combining Efficiency with Precision. In *6th Static Analysis Symposium – SAS ’99 (Lecture Notes in Computer Science, Vol. 1694)*, Agostino Cortesi and Gilberto Filé (Eds.). Springer, 232–247. https://doi.org/10.1007/3-540-48294-6_15
- 1395 Davide Sangiorgi. 2009. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.* 31, 4 (2009), 15:1–15:41. <https://doi.org/10.1145/1516507.1516510>
- 1396 Sigurd Schneider. 2013. *Semantics of an Intermediate Language for Program Transformation*. Master’s thesis. Saarland University. <https://www.ps.uni-saarland.de/~sdschn/msc.pdf>
- 1397 Vugranam C. Sreedhar and Guang R. Gao. 1995. A Linear Time Algorithm for Placing phi-nodes. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1995)*, Ron K. Cytron and Peter Lee (Eds.). 62–73. <https://doi.org/10.1145/199448.199464>
- 1398 James Stanier. 2016. *Graphs and Gating Functions* (1st ed.), Chapter 14. In [Rastello 2016].

1420
1421

- 1422 André Luiz Camargos Tavares, Benoit Boissinot, Fernando Magno Quintão Pereira, and Fabrice Rastello. 2014. Parameterized
 1423 Construction of Program Representations for Sparse Dataflow Analyses. In *23rd International Conference on Compiler*
 1424 *Construction – CC 2014 (Lecture Notes in Computer Science, Vol. 8409)*, Albert Cohen (Ed.). Springer, 18–39. https://doi.org/10.1007/978-3-642-54807-9_2
- 1425 Michael James Van Emmerik. 2007. *Static single assignment for decompilation*. Ph.D. Dissertation. University of Queensland.
- 1426 Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant Propagation with Conditional Branches. *ACM Trans. Program.*
 1427 *Lang. Syst.* 13, 2 (1991), 181–210. <https://doi.org/10.1145/103135.103136>
- 1428 Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast
 1429 and extensible equality saturation. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29. <https://doi.org/10.1145/3434304>
- 1430 S. Bharadwaj Yadavalli and Aaron Smith. 2019. Raising binaries to LLVM IR with MCTOLL (WIP paper). In *20th ACM*
 1431 *SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2019)*, Jian-Jia
 1432 Chen and Aviral Shrivastava (Eds.). 213–218. <https://doi.org/10.1145/3316482.3326354>
- 1433 Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *32nd*
 1434 *ACM SIGPLAN Conference on Programming Language Design and Implementation*, :SERIES: PLDI 2011, Mary W. Hall and
 1435 David A. Padua (Eds.). ACM, 283–294. <https://doi.org/10.1145/1993498.1993532>
- 1436 Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM intermediate rep-
 1437 resentation for verified program transformations. In *39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming*
 1438 *Languages (POPL 2012)*, John Field and Michael Hicks (Eds.). 427–440. <https://doi.org/10.1145/2103656.2103709>

1438 A SOUNDNESS OF THE SYMBOLIC EXPRESSION ANALYSIS

1439 While the description of the symbolic expression analysis (performing global value numbering)
 1440 is sufficiently simple that its complete description fits in Figure 4, the soundness proof is more
 1441 complex. This soundness proof is interesting in that it shows that the interpretation of instruction is
 1442 very precise, and that approximations only comes from the join operation (for which the soundness
 1443 is more involved).

1443 A.1 Soundness and completeness of abstract evaluation of instructions

1444 For the analysis of instructions we prove a stronger version of the usual soundness theorem, where
 1445 the argument and result of transfer functions are related using the same valuation Γ .

1446 LEMMA A.1 (STRONG SOUNDNESS AND COMPLETENESS OF $\mathcal{E}^\# \llbracket \cdot \rrbracket ()$). *Let $\sigma^\# \in \Sigma^\#$, e an expression.*
 1447 *Let $\Gamma \mapsto \sigma \in \gamma_{\Sigma^\#}(\sigma^\#)$, i.e. Γ and σ are such that $\sigma = \gamma_{\Sigma^\#}(\sigma^\#)[\Gamma]$. Then:*

$$1448 \mathcal{E} \llbracket e \rrbracket (\sigma) = \mathcal{E} \llbracket \mathcal{E}^\# \llbracket e \rrbracket (\sigma^\#) \rrbracket (\Gamma)$$

1449 PROOF. By structural induction on e , using the hypothesis that $\forall x, \sigma[x] = \mathcal{E} \llbracket \sigma^\#[x] \rrbracket ()$ for
 1450 the base variable case. \square

1451 Example A.2. Let $\sigma^\# = [x \mapsto (2+x_{\ell_0})]$, $\Gamma = [x_{\ell_0} \mapsto 7]$, and $\sigma = [x \mapsto 9]$. We have $\sigma = \gamma_{\Sigma^\#}(\sigma^\#)[\Gamma]$.

- 1452 • $\mathcal{E}^\# \llbracket 1 + x \rrbracket (\sigma^\#) = \text{subst}(1 + x, \sigma^\#) = 1 + (2 + x_{\ell_0})$. We have $\mathcal{E} \llbracket 1 + (2 + x_{\ell_0}) \rrbracket (\Gamma) = 10$.
- 1453 • $\mathcal{E}^\# \llbracket 1 + x \rrbracket (\sigma) = 1 + 9 = 10$.

1454 LEMMA A.3 (STRONG SOUNDNESS OF $\mathcal{I}^\# \llbracket \cdot \rrbracket ()$). *Let $\sigma^\# \in \Sigma^\#$, i an instruction. Let $\Gamma \mapsto \sigma \in \gamma_{\Sigma^\#}(\sigma^\#)$,*
 1455 *i.e. Γ and σ are such that $\sigma = \gamma_{\Sigma^\#}(\sigma^\#)[\Gamma]$. Then:*

$$1456 \mathcal{I} \llbracket i \rrbracket (\sigma) = \mathcal{I}^\# \llbracket i \rrbracket (\sigma^\#)[\Gamma]$$

1457 PROOF. Both $\mathcal{I} \llbracket \cdot \rrbracket ()$ and $\mathcal{I}^\# \llbracket \cdot \rrbracket ()$ define sequence of instructions as function composition, so we
 1458 only need to consider the case where i is an assignment. We have $\mathcal{I} \llbracket [x := e] \rrbracket (\sigma) = \sigma[x \leftarrow \mathcal{E} \llbracket e \rrbracket (\sigma)]$
 1459 and $\mathcal{I}^\# \llbracket [x := e] \rrbracket (\sigma^\#) = \sigma[x \leftarrow \mathcal{E}^\# \llbracket e \rrbracket (\sigma^\#)]$, so we can conclude the proof using Lemma A.1. \square

1460 COROLLARY A.4 (SOUNDNESS AND COMPLETENESS OF $\mathcal{I}^\# \llbracket \cdot \rrbracket ()$). *Let i an instruction and $\sigma^\# \in \Sigma^\#$.*

$$1461 \gamma_{\Sigma^\#} \circ \mathcal{I}^\# \llbracket i \rrbracket (\sigma^\#) = \{ \Gamma \mapsto \mathcal{I} \llbracket i \rrbracket (\sigma) \mid \Gamma \mapsto \sigma \in \gamma_{\Sigma^\#}(\sigma^\#) \}$$

PROOF. Immediate consequence of Lemma A.3. \square

A.2 Soundness of \sqcup^ℓ

The \sqcup^ℓ operator is not sound when applied to any pair of elements in Σ^\sharp . The problem comes from the fact that we reuse variable identifiers, which could thus create spurious relations between terms (this problem does not happen in analyses which create fresh variables instead). Consider for instance:

$$[x \mapsto 1 + y_{\ell_6}, y \mapsto 2] \sqcup^{\ell_6} [x \mapsto 1 + y_{\ell_6}, y \mapsto 3] = [x \mapsto 1 + y_{\ell_6}, y \mapsto y_{\ell_6}]$$

which contains a relation between x and y through the shared use of y_{ℓ_6} , while this relation does not hold in the concrete. We can thus only prove a weaker soundness theorem:

Definition A.5 (Occurs). Let $p^\sharp \in \mathbb{L} \rightarrow \Sigma_\perp^\sharp$. Given two locations ℓ and ℓ_D , we say that ℓ_D occurs in $p^\sharp[\ell]$ if a variable v exists, with $\text{loc}(v) = \ell_D$ and v occurs in $p^\sharp[\ell]$ (i.e. v is a subterm of $p^\sharp[\ell][x]$ for some $x \in \mathbb{X}$). For instance, if $p^\sharp = [\ell_8 \mapsto [x \mapsto 1 + x_{\ell_3}]]$, then ℓ_3 occurs in $p^\sharp[\ell_8]$.

LEMMA A.6 (SOUNDNESS OF \sqcup^ℓ). Let σ_1^\sharp and $\sigma_2^\sharp \in \Sigma^\sharp$. If ℓ does not occur inside one of σ_1^\sharp or σ_2^\sharp , then

$$\gamma_{\Sigma^\sharp}(\sigma_1^\sharp \sqcup^\ell \sigma_2^\sharp) \supseteq \gamma_{\Sigma^\sharp}(\sigma_1^\sharp) \cup \gamma_{\Sigma^\sharp}(\sigma_2^\sharp)$$

PROOF. Let $\sigma^\sharp = \sigma_1^\sharp \sqcup^\ell \sigma_2^\sharp$. Let $\Gamma_1 \in \Gamma$ and $\sigma_1 = \gamma_{\Sigma^\sharp}(\sigma_1^\sharp)[\Gamma_1]$. We show that Γ exists, such that $\sigma_1 = \gamma_{\Sigma^\sharp}(\sigma^\sharp)[\Gamma]$. For this we take $\Gamma = \left[v \in \mathbb{V} \mapsto \begin{cases} \sigma_1[x] & \text{if } \exists x \in \mathbb{X} : v = x_\ell \\ \Gamma_1[v] & \text{otherwise} \end{cases} \right]$.

- For variables x such that $\sigma_1^\sharp[x] \neq \sigma_2^\sharp[x]$, i.e. such that $\sigma^\sharp[x] = x_\ell$, it is clear that $\gamma_{\Sigma^\sharp}(\sigma^\sharp)[\Gamma][x] = \mathcal{E}[x_\ell](\Gamma) = \sigma_1[x]$.
- For variables x such that $\sigma^\sharp[x] = \sigma_1^\sharp[x] = \sigma_2^\sharp[x]$, we note that ℓ does not appear in any subterm of $\sigma^\sharp[x]$ by hypothesis. Thus, $\gamma_{\Sigma^\sharp}(\sigma^\sharp)[\Gamma][x] = \gamma_{\Sigma^\sharp}(\sigma_1^\sharp)[\Gamma_1][x]$ (as Γ and Γ_1 have the same values for variables whose location is not ℓ). And because $\gamma_{\Sigma^\sharp}(\sigma_1^\sharp)[\Gamma_1][x] = \sigma_1[x]$, this allows to conclude that $\gamma_{\Sigma^\sharp}(\sigma^\sharp)[\Gamma][x] = \sigma_1[x]$.

A.3 Soundness of $\mathcal{F}_{\mathbb{L} \rightarrow \Sigma_\perp^\sharp}$

Fortunately, $\mathcal{F}_{\mathbb{L} \rightarrow \Sigma_\perp^\sharp}$ will always use \sqcup^ℓ in the context where it is sound. This is formally established using the following definition and theorem:

Definition A.7 (Location of a variable). Let $x_\ell \in \mathbb{V}$. We call ℓ the location of the variable x_ℓ , and we define the function $\text{loc}(\cdot) : \mathbb{V} \rightarrow \mathbb{L}$ the function that returns the location of its argument (i.e. $\text{loc}(x_\ell) = \ell$).

LEMMA A.8. Let $n \in \mathbb{N}$ and $p_n^\sharp = \mathcal{F}_{\mathbb{L} \rightarrow \Sigma_\perp^\sharp}^{(n)}([\!])$. Then, for all $\ell \in \mathbb{L}$, for any location ℓ' on any simple (acyclic) path from ℓ_0 to ℓ , ℓ does not occur in ℓ' .

PROOF. The proof relies on the property (P) that if $n > 0$ and $p_n^\sharp[\ell] \neq \perp$, then there exists a simple path from ℓ_0 to a predecessor of ℓ such that every element ℓ' in that path is such that $p_{n-1}^\sharp[\ell'] \neq \perp$, which is easily proved by induction.

The main proof is then done by induction of the property of the Lemma on n .

- the property is obviously true for $n = 0$, as $\mathcal{F}_{\mathbb{L} \rightarrow \Sigma_\perp^\sharp}^{(0)}([\!]) = [\!]$, whose domain is empty.

- when $n > 0$, we note $p_{n-1}^\# = \mathcal{F}_{\perp \rightarrow \Sigma_\perp^\#}^{(n-1)}(\perp)$ and $p_n^\# = \mathcal{F}_{\perp \rightarrow \Sigma_\perp^\#}(p_{n-1}^\#)$.
 - the property is true for ℓ_0 by definition (there is no other location in the path from ℓ_0 to ℓ_0).
 - we consider $\ell \in \perp$ and ℓ' on a simple path from ℓ_0 to ℓ . By definition of $\mathcal{F}_{\perp \rightarrow \Sigma_\perp^\#}$, we see that ℓ occurs in $p_n^\#[\ell']$ only if for all the predecessors ℓ'' of ℓ' , ℓ occurs in $p_{n-1}^\#[\ell'']$. But either $p_n^\#[\ell'] = \perp$, and then trivially ℓ does not occur in $p_n^\#[\ell']$. Or $p_n^\#[\ell'] \neq \perp$, and then by (P) there is a predecessor ℓ'' of ℓ' such that $p_{n-1}^\#[\ell''] \neq \perp$, and further ℓ'' is on a simple path from ℓ_0 to ℓ' . Using the induction hypothesis, we know that ℓ does not occur in $p_{n-1}^\#[\ell'']$. This concludes that ℓ does not occur in $p_n^\#[\ell']$.

□

With this lemma, we can now proceed to prove the soundness of the analysis:

THEOREM A.9 (SOUNDNESS OF $\mathcal{F}_{\perp \rightarrow \Sigma_\perp^\#}$).

$$\forall p^\# \in \perp \rightarrow \Sigma_\perp^\# : (\mathcal{F}_{\mathcal{P}(\mathbb{S})} \circ \gamma_{\perp \rightarrow \Sigma_\perp^\#})(p^\#) \subseteq (\gamma_{\perp \rightarrow \Sigma_\perp^\#} \circ \mathcal{F}_{\perp \rightarrow \Sigma_\perp^\#})(p^\#)$$

PROOF. We focus on the computation of $p^\# = \mathcal{F}_{\perp \rightarrow \Sigma_\perp^\#}(p^\#)$ at location ℓ . Thanks to Lemma A.8, we know that for at least one of the predecessors ℓ' of ℓ , the location ℓ will not occur in $p^\#[\ell']$. This will still be the case of $\mathcal{I}^\#[\text{instr}[\ell']](p^\#[\ell'])$, by definition. Thus, the usage of \sqcup^ℓ will be sound in this case. By combination with the soundness of $\mathcal{I}^\#\cdot$, we can conclude on the soundness of $\mathcal{F}_{\perp \rightarrow \Sigma_\perp^\#}$. □

Note. This analysis is sound, but it is not exact: notably, tests are not taken into account, nor is the semantic of the binary operation (we do as if they were uninterpreted functions); finally, the creation of simultaneous variables loose any relations that may exist between these variables. It is obviously usual for abstract domains to lose information; but it is still interesting to note that despite the fact that global value numbering is not exact, the SSA abstraction that builds on it will be exact.

A.4 Soundness of the symbolic expression analysis

Our main soundness theorem for the global value numbering states that the analysis computes a sound overapproximation of all the reachable states of the program.

THEOREM A.10 (SOUNDNESS OF GLOBAL VALUE NUMBERING). *Let $p^\#$ be a fixpoint of $\mathcal{F}_{\perp \rightarrow \Sigma_\perp^\#}$, i.e. $p^\# = \mathcal{F}_{\perp \rightarrow \Sigma_\perp^\#}(p^\#)$. Then $p^\#$ is a sound overapproximation of the reachable states of the $(\mathbb{S}, \rightsquigarrow)$ transition system:*

$$\gamma_{\perp \rightarrow \Sigma_\perp^\#}(p^\#) \supseteq \mathbb{S}_0^*$$

PROOF. By Theorem A.9 and the Tarski fixpoint transfer theorem. □

Note. The soundness of the algorithm works independently of the iteration strategy being used; we are thus free to use chaotic iteration methods [Cousot 1977] instead of the naive method presented in this formalization, to compute the global value numbering.

B PROOFS OF SCOPE AND DOMINANCE PROPERTIES

In this section, let p^\sharp be a fixpoint of $\mathcal{F}_{\mathbb{L} \rightarrow \Sigma^\sharp}$ in a program whose graph is \mathcal{G} . We note by $\text{freevars}(e)$ the set of all the variables $v \in \mathbb{V}$ that occur in e .

THEOREM 5.1 (STRICTNESS). $\forall \ell \in \mathbb{L}, x \in \mathbb{X}, v \in \mathbb{V} : v \text{ occurs in } p^\sharp[\ell][x] \Rightarrow \text{loc}(v) \text{ dominates } \ell$.

PROOF OF THEOREM 5.1. We can prove the following property:

$$\forall \ell_D, \ell \in \mathbb{L} : \ell_D \text{ occurs in } p^\sharp[\ell] \Rightarrow \ell = \ell_D \text{ or } \forall \ell' \in \text{preds}(\ell) : \ell_D \text{ occurs in } p^\sharp[\ell'] \quad (2)$$

This is proved from the definition of $\mathcal{F}_{\mathbb{L} \rightarrow \Sigma^\sharp}$: when analyzing location ℓ , if $\ell = \ell_D$, a new symbolic variable v with $\text{loc}(v) = \ell_D$ can be created. If $\ell \neq \ell_D$, assume there is v with $\text{loc}(v) = \ell_D$ that occurs in $p^\sharp[\ell]$. Then v must be present in all the predecessors: the \sqcup^ℓ operation replaces the term with a new symbolic variable (with a location ℓ) when the abstract stores differ, so v must occur in all arguments to \sqcup^ℓ . Because $\mathcal{I}^\sharp[\llbracket \cdot \rrbracket](\cdot)$ cannot introduce new variables, v must occur in $p^\sharp[\ell']$ for all the predecessors ℓ' or ℓ .

Now, iterating (2) implies that every path arriving at ℓ must either come from ℓ_D , or from a location with no predecessor. The only location with no predecessor is ℓ_0 , and ℓ_D does not occur in the abstract store at ℓ_0 . Hence, ℓ_D dominates ℓ . \square

THEOREM B.1. *If both ℓ_1 and ℓ_2 dominate ℓ , then either ℓ_1 dominate ℓ_2 or ℓ_2 dominate ℓ_1 .*

PROOF OF THEOREM B.1. The proof can be found in [Appel 1998a, p.408]. \square

THEOREM 5.6. $\text{scope}_{\mathcal{G}}(e) = \{\ell \mid \overline{\text{scope}}_{\mathcal{G}}(e) \text{ dominates } \ell\}$

PROOF OF THEOREM 5.6. We prove this property by induction on e :

- Case $e = v$: $\text{scope}_{\mathcal{G}}(v) = \{\ell \mid \overline{\text{scope}}_{\mathcal{G}}(v) \text{ dominates } \ell\}$ (by definition)
- Case $e = z$: $\text{scope}_{\mathcal{G}}(z) = \mathbb{L} = \{\ell \mid \ell_0 \text{ dominates } \ell\}$ (because \mathcal{G} only contains reachable locations and execution starts at ℓ_0)
- Case $e = e_1 \diamond e_2$: Note that our induction property can be equivalently formulated as $\forall \ell \in \mathbb{L} : \ell \in \text{scope}_{\mathcal{G}}(e) \Leftrightarrow \ell \text{ dominated by } \overline{\text{scope}}_{\mathcal{G}}(e)$. And:

$$\begin{aligned} \ell \in \text{scope}_{\mathcal{G}}(e_1 \diamond e_2) &\Leftrightarrow \ell \in \text{scope}_{\mathcal{G}}(e_1) \wedge \ell \in \text{scope}_{\mathcal{G}}(e_2) && \text{(by definition)} \\ &\Leftrightarrow \ell \text{ dominated by } \overline{\text{scope}}_{\mathcal{G}}(e_1) \wedge \ell \text{ dominated by } \overline{\text{scope}}_{\mathcal{G}}(e_2) && \text{(recursion hypothesis)} \\ &\Leftrightarrow \ell \text{ dominated by } \min(\overline{\text{scope}}_{\mathcal{G}}(e_1), \overline{\text{scope}}_{\mathcal{G}}(e_2)) && \text{(using Theorem B.1)} \\ &\Leftrightarrow \ell \text{ dominated by } \overline{\text{scope}}_{\mathcal{G}}(e_1 \diamond e_2) && \text{(by definition)} \end{aligned}$$

\square

C SOUNDNESS AND COMPLETENESS OF THE SSA ABSTRACTION

C.1 \mathcal{H}_{p^\sharp} is a strong homomorphism

We now prove that when p^\sharp, \hat{g} is a fixpoint, \mathcal{H}_{p^\sharp} is a strong homomorphism from $\rightsquigarrow_{\hat{g}}$ to \rightsquigarrow .

This result is linked to the soundness (and completeness) of the $\mathcal{F}_{\text{SSA}^\sharp}$ transfer function, that we prove in the case where we reached a fixpoint p^\sharp, \hat{g} :

LEMMA C.1 (SOUNDNESS AND COMPLETENESS OF $\mathcal{F}_{\text{SSA}^\sharp}$). *Let (p^\sharp, \hat{g}) such that $\mathcal{F}_{\text{SSA}^\sharp}(p^\sharp, \hat{g}) = (p^\sharp, \hat{g})$. Let $\ell' \in \mathbb{L}$ and $\ell \in \text{preds}(\ell')$. Let $\sigma^\sharp = p^\sharp[\ell]$ and $\sigma'^\sharp = p^\sharp[\ell']$. Let $\Gamma \mapsto \sigma \in \mathcal{Y}_{\Sigma^\sharp}(\sigma^\sharp)$, i.e. $\sigma = \mathcal{Y}_{\Sigma^\sharp}(\sigma^\sharp)[\Gamma]$.*

Then:

$$\{\sigma' : (\ell, \sigma) \mapsto (\ell', \sigma')\} = \{\gamma_{\Sigma^\#}(\sigma'^\#)[\Gamma'] : (\ell, \Gamma) \rightsquigarrow_{\hat{g}} (\ell', \Gamma')\} \quad (3)$$

$$\{(\ell', \sigma') : (\ell, \sigma) \mapsto (\ell', \sigma')\} = \{\mathcal{H}_{p^\#}(\ell', \Gamma') : (\ell, \Gamma) \rightsquigarrow_{\hat{g}} (\ell', \Gamma')\} \quad (4)$$

PROOF. Equation (4) is obtained from (3) by expanding the definition of $\mathcal{H}_{p^\#}$, so we can concentrate on the proof of (3). The proof is simplified by the fact that both transitions $\rightsquigarrow_{g^\#}$ and \mapsto are deterministic when the next location is fixed. More precisely, let \hat{e}, B be such that $(\ell, \ell') \mapsto (\hat{e}, B) \in \hat{g}$, then

$$\begin{aligned} \exists \sigma' : (\ell, \sigma) \mapsto (\ell', \sigma') &\Rightarrow \sigma' = I[\text{instr}[\ell]](\sigma) \\ \exists \Gamma' : (\ell, \Gamma) \rightsquigarrow_{\hat{g}} (\ell', \Gamma') &\Rightarrow \Gamma' = \text{unbind}_{\hat{g}}(\ell', \text{bind}(B, \Gamma)) \end{aligned}$$

Thus the set $\{\sigma' : (\ell, \sigma) \mapsto (\ell', \sigma')\}$ is either a singleton or the empty set, and the same is true for $\{\gamma_{\Sigma^\#}(\sigma'^\#)[\Gamma'] : (\ell, \Gamma) \rightsquigarrow_{\hat{g}} (\ell', \Gamma')\}$.

The condition for the former set to be non-empty is that if e is the condition guarding the transition from ℓ to ℓ' (i.e. $\mathcal{G}[\ell, \ell'] = e$), then $\mathcal{E}[e](\sigma') \neq 0$. Similarly, $(\ell, \Gamma) \rightsquigarrow_{g^\#} (\ell', \Gamma')$ if, and only if, $\mathcal{E}[\hat{e}](\Gamma) \neq 0$. Using Lemma A.1 (which says that the abstract evaluation of expression $\mathcal{E}^\#\cdot$ is exact) and the definition of “edge”, we know that both conditions are equivalent. In the case where there is no edge from ℓ to ℓ' in \hat{g} , by definition of edge it means that either there is no edge from ℓ to ℓ' in \mathcal{G} , or that this edge has 0 for guard, or that $p^\#[\ell] = \perp$ and there is no state that reaches ℓ : in every case the left and right sets are empty.

It remains to show that $\sigma' = \gamma_{\Sigma^\#}(\sigma'^\#)[\Gamma']$. The main idea is that the binding B produced by the edge function binds all the variables that are introduced at ℓ' . Combined with Lemma A.3 (saying that the abstract evaluation of instructions $\mathcal{I}^\#\cdot$ is exact), this allows to prove that $\sigma' = \gamma_{\Sigma^\#}(\sigma'^\#)[\text{bind}(B, \Gamma)]$.

However, Γ' is not just $\text{bind}(B, \Gamma)$, but $\text{unbind}_{\hat{g}}(\ell', \text{bind}(B, \Gamma))$. It remains to prove that this unbind operation has no effect. This is achieved using Theorem 5.3 (and the fact that $\text{scope}_{\mathcal{G}} = \text{scope}_{\hat{g}}$), that says that because $p^\#$ is a fixed point, only symbolic variables that are scoped in ℓ' will occur in $\sigma'^\#$, thus unbind has no effect on the evaluation. Thus:

$$\gamma_{\Sigma^\#}(\sigma'^\#)(\Gamma') = \gamma_{\Sigma^\#}(\sigma'^\#)[\text{unbind}_{\hat{g}}(\ell', \text{bind}(B, \Gamma))] = \gamma_{\Sigma^\#}(\sigma'^\#)[\text{bind}(B, \Gamma)] = \sigma'.$$

□

One consequence of this Lemma is the following theorem.

THEOREM C.2 (SAME SUCCESSOR STATES). *Let $(p^\#, \hat{g})$ such that $\mathcal{F}_{\text{SSA}^\#}(p^\#, \hat{g}) = (p^\#, \hat{g})$. Let $s \in \mathbb{S}, \hat{s} \in \hat{\mathbb{S}}$ such that $s = \mathcal{H}_{p^\#}(\hat{s})$. Then:*

$$\{s' \in \mathbb{S} \mid s \mapsto s'\} = \{\mathcal{H}_{p^\#}(\hat{s}') \mid \hat{s} \rightsquigarrow_{\hat{g}} \hat{s}'\} \quad (5)$$

PROOF. Obtained from Lemma C.1 (equation (4)) by union over all the successors of the location of s . □

COROLLARY C.3. $\mathcal{H}_{p^\#}$ is a strong homomorphism.

PROOF. Equation (5) is equivalent to Equation (1). □

A consequence of $\mathcal{H}_{p^\#}$ being a strong homomorphism is that it is a homomorphism, and thus a simulation, which implies that the traces of $(\hat{\mathbb{S}}, \rightsquigarrow_{\hat{g}})$ are included in the traces of the (\mathbb{S}, \mapsto) transition system. Thus, we can conclude that $(p^\#, \hat{g})$ represents a subset of all the traces of (\mathbb{S}, \mapsto) (and is thus complete). In the next section, we also prove the soundness.

C.2 $\mathcal{H}_{p^\#}$ is a surjective strong homomorphism between the reachable states

Up to now we could conclude that $\mathcal{H}_{p^\#}$ is a simulation from SSA to the original program, but we did not prove that it is a simulation from the original program to SSA. And indeed it is not: there are transitions in the original transition system that cannot be simulated in the SSA transition system.

Example C.4. Taking the transition system stemming from the program of Figure 1b, we have for instance:

$$\text{let } s = \left(\ell_3, \begin{bmatrix} \text{inc} & \mapsto -4 \\ \text{time} & \mapsto 35 \\ \text{i} & \mapsto 10 \\ \text{hours} & \mapsto 0 \end{bmatrix} \right) \text{ and } s' = \left(\ell_4, \begin{bmatrix} \text{inc} & \mapsto -4 \\ \text{time} & \mapsto 31 \\ \text{i} & \mapsto 11 \\ \text{hours} & \mapsto 0 \end{bmatrix} \right). \text{ Then: } s \mapsto s'.$$

This $s \mapsto s'$ transition cannot be simulated by the SSA program, because the states where $\text{inc} \neq 1$ and $\text{time} \neq \text{i}$ cannot be represented in the SSA version (i.e. : $\forall \hat{s} \in \hat{\mathbb{S}} : \mathcal{H}_{p^\#}(\hat{s}) \neq s$).

However, the reason why states like s or s' cannot be represented by $p^\#$ is that they are not reachable: $p^\#$ is a sound abstraction of the reachable states \mathbb{S}_0^* (Theorem A.10), but nothing requires that $p^\#$ represent non-reachable states.

We thus want to restrict the transition relation \mapsto to the reachable states:

$$\mapsto_{|\mathbb{S}^*} \triangleq \{(s, s') \in \mathbb{S}^* \mid s \mapsto s'\} \quad (6)$$

However, $\mathcal{H}_{p^\#}$ is no longer an homomorphism from $\rightsquigarrow_{\hat{g}}$ to $\mapsto_{|\mathbb{S}^*}$, because unreachable SSA transitions cannot be matched in $\mapsto_{|\mathbb{S}^*}$.

Example C.5. Indeed, if we take again Figure 1 as an example:

$$\text{let } \hat{s} = \left(\ell_1, \begin{bmatrix} \text{inc}_{\ell_0} & \mapsto 11 \\ \text{time}_{\ell_0} & \mapsto 22 \\ \text{i}_{\ell_0} & \mapsto 33 \\ \text{hours}_{\ell_0} & \mapsto 44 \\ \text{time}_i & \mapsto -1 \\ \text{hours}_i & \mapsto 27 \end{bmatrix} \right) \text{ and } \hat{s}' = \left(\ell_2, \begin{bmatrix} \text{inc}_{\ell_0} & \mapsto 11 \\ \text{time}_{\ell_0} & \mapsto 22 \\ \text{i}_{\ell_0} & \mapsto 33 \\ \text{hours}_{\ell_0} & \mapsto 44 \\ \text{time}_i & \mapsto -1 \\ \text{hours}_i & \mapsto 27 \end{bmatrix} \right)$$

Then we have $\hat{s} \rightsquigarrow_{\hat{g}} \hat{s}'$ but we do not have $\mathcal{H}_{p^\#}(\hat{s}) \mapsto_{|\mathbb{S}^*} \mathcal{H}_{p^\#}(\hat{s}')$, because $\mathcal{H}_{p^\#}(\hat{s})$ is not a reachable state.

We thus also need to restrict SSA transitions to the reachable SSA states.

Definition C.6 (Reachable SSA state). Given \hat{g} , we define $\hat{\mathbb{S}}^*$ to be the reachable set of SSA states:

$$\hat{\mathbb{S}}^* \triangleq \{\hat{s} \in \hat{\mathbb{S}} : \exists \hat{s}_0 \in \hat{\mathbb{S}}_0, \hat{s}_0 \rightsquigarrow_{\hat{g}}^* \hat{s}\}$$

where $\rightsquigarrow_{\hat{g}}^*$ represent the reflexive and transitive closure of $\rightsquigarrow_{\hat{g}}$.

Definition C.7 (Restricted SSA transition).

$$\rightsquigarrow_{\hat{g}|\hat{\mathbb{S}}^*} \triangleq \{(\hat{s}, \hat{s}') \in \hat{\mathbb{S}}^* \mid \hat{s} \rightsquigarrow_{\hat{g}} \hat{s}'\} \quad (7)$$

We can now prove that $\mathcal{H}_{p^\#}$ is a surjective strong homomorphism on these restricted relations:

THEOREM C.8 ($\mathcal{H}_{p^\#}$ IS A SURJECTIVE STRONG HOMOMORPHISM). $\mathcal{H}_{p^\#}$ is a strong homomorphism from $\rightsquigarrow_{\hat{g}|\hat{\mathbb{S}}^*}$ to $\mapsto_{|\mathbb{S}^*}$:

$$\forall \hat{s}, \hat{s}' \in \hat{\mathbb{S}} : \hat{s} \rightsquigarrow_{\hat{g}|\hat{\mathbb{S}}^*} \hat{s}' \Leftrightarrow \mathcal{H}(\hat{s}) \mapsto_{|\mathbb{S}^*} \mathcal{H}(\hat{s}') \quad (8)$$

$\mathcal{H}_{p^\#}$ is surjective on the field of $\succrightarrow_{|\mathbb{S}^*}$:

$$\forall s, s' \in \mathbb{S} : \mathcal{H}(\hat{s}) \succrightarrow_{|\mathbb{S}^*} \mathcal{H}(\hat{s}') \Rightarrow \exists \hat{s}, \hat{s}' \in \hat{\mathbb{S}} : s = \mathcal{H}(\hat{s}) \wedge s' = \mathcal{H}(\hat{s}') \quad (9)$$

PROOF. We note by \mathbb{S}^n and $\hat{\mathbb{S}}^n$ respectively the original program states and SSA states reachable after n transitions, and we show by induction on n that

$$\forall \hat{s}, \hat{s}' \in \hat{\mathbb{S}}^n : \hat{s} \rightsquigarrow_{g|\hat{\mathbb{S}}^*} \hat{s}' \Leftrightarrow \mathcal{H}(\hat{s}) \succrightarrow_{|\mathbb{S}^*} \mathcal{H}(\hat{s}')$$

using Theorem C.2. □

Note that it is quite clear that a surjective strong homomorphism is also a bisimulation.

C.3 Meaning, soundness and completeness of the SSA abstraction

Up to now we discussed how an SSA abstraction can be constructed using transfer functions, and the strong relation between the SSA abstraction and the original program. What is missing is to define the concretization for the SSA abstraction.

Definition C.9 (Concretization of the SSA abstraction). We define the concretization of the $\text{SSA}^\#$ domain as follows:

$$\gamma_{\text{SSA}^\#} : \text{SSA}^\# \rightarrow \mathcal{P}(\mathcal{P}(\mathbb{S} \times \mathbb{S}))$$

$$\gamma_{\text{SSA}^\#}(p^\#, \hat{g}) = \{(\mathcal{H}_{p^\#}(\hat{s}), \mathcal{H}_{p^\#}(\hat{s}')) \mid \hat{s} \rightsquigarrow_{g|\hat{\mathbb{S}}^*} \hat{s}'\}$$

(note that the definition of $\hat{\mathbb{S}}^*$ depends on \rightsquigarrow_g).

Using this concretization, we can say that what our SSA-transformation analysis computes a lossless (but non-trivial) abstraction of the transition system of the original program.

THEOREM C.10 (SOUNDNESS AND COMPLETENESS OF THE SSA ABSTRACTION COMPUTATION).

Let $(p^\#, \hat{g})$ be a fixpoint of $\mathcal{F}_{\text{SSA}^\#}$. Then: $\gamma_{\text{SSA}^\#}(p^\#, \hat{g}) = \succrightarrow_{|\mathbb{S}^*}$

PROOF. By definition of $\gamma_{\text{SSA}^\#}(p^\#, \hat{g})$ and Theorem C.8. □

Following the hierarchy of semantics [Cousot 2002], we can derive from this result that our analysis also computes weaker abstractions of the original program in a lossless way. For instance, the result of our analysis also exactly represents the set of reachable states:

THEOREM C.11 (SOUNDNESS AND COMPLETENESS OF THE SSA ABSTRACTION COMPUTATION VIEWED AS REACHABLE STATES). Let γ be defined as

$$\gamma : \text{SSA}^\# \rightarrow \mathcal{P}(\mathbb{S})$$

$$\gamma(p^\#, \hat{g}) = \{s \in \mathbb{S} : \exists s_0 \in \mathbb{S}_0, s_0 \mathcal{R}^* s\} \text{ where } \mathcal{R} \triangleq \gamma_{\text{SSA}^\#}(p^\#, \hat{g})$$

Let $(p^\#, \hat{g})$ be a fixpoint of $\mathcal{F}_{\text{SSA}^\#}$. Then:

$$\gamma(p^\#, \hat{g}) = \mathbb{S}^*$$

PROOF. Consequence of Theorem C.10 □

The same could be done for the set of program traces, or the set of pairs of input/output states of the program. When viewed as a program transformation, this result also proved that the SSA translation preserved the semantics of the original program.

D PROOFS FOR INCREMENTALLY COMPLETE CONSTRUCTION OF SSA

D.1 Soundness and completeness of the analysis when a post-fixpoint is reached

LEMMA D.1. *The following conditions are equivalent:*

$$\sigma_1^\# \sqsubseteq^\ell \sigma_2^\# = \sigma_2^\# = (\sigma_1^\# \sqcup^\ell \sigma_2^\#) \quad (10)$$

$$\sigma_1^\# \sqsubseteq^\ell \sigma_2^\# = \exists \sigma_3^\# : \sigma_2^\# = \sigma_1^\# \sqcup^\ell \sigma_3^\# \quad (11)$$

PROOF. • (10) \Rightarrow (11): it suffices to pick $\sigma_3^\# = \sigma_2^\#$.

• (11) \Rightarrow (10):

$$\begin{aligned} (\sigma_1^\# \sqcup^\ell \sigma_2^\#) &= \sigma_1^\# \sqcup^\ell (\sigma_1^\# \sqcup^\ell \sigma_3^\#) && \text{(by definition of } \sigma_2^\# \text{)} \\ &= (\sigma_1^\# \sqcup^\ell \sigma_1^\#) \sqcup^\ell \sigma_3^\# && \text{(associativity of } \sqcup^\ell \text{)} \\ &= \sigma_1^\# \sqcup^\ell \sigma_3^\# && \text{(idempotence of } \sqcup^\ell \text{)} \\ &= \sigma_2^\# && \text{(by definition of } \sigma_2^\# \text{)} \end{aligned}$$

□

Definition D.2 (Order relation on $\mathbb{L} \rightarrow \Sigma_\perp^\#$). Let $p_1^\#, p_2^\# \in \mathbb{L} \rightarrow \Sigma_\perp^\#$. We define:

$$p_1^\# \sqsubseteq p_2^\# \triangleq \forall \ell \in \mathbb{L} : p_1^\#[\ell] \sqsubseteq^\ell p_2^\#[\ell]$$

Definition D.3 (Post-fixpoint). $p^\# \in \mathbb{L} \rightarrow \Sigma_\perp^\#$ is a *post-fixpoint* of $\mathcal{F}_{\mathbb{L} \rightarrow \Sigma_\perp^\#}$ if: $\mathcal{F}_{\mathbb{L} \rightarrow \Sigma_\perp^\#}(p^\#) \sqsubseteq p^\#$.

This is the classic definition of a post-fixpoint in abstract interpretation.

Now, many of the previous theorems that applied when $p^\#$ was a fixpoint of the program can be generalized to the case where $p^\#$ is only a post-fixpoint (see Theorems D.4, D.5 and Lemma D.6 in the appendix). In particular, we can generalize our soundness theorem to the post-fixpoint case:

THEOREM D.4 (THEOREM 5.1 IN THE POST-FIXPOINT CASE). *Let $p^\#$ be a post-fixpoint of $\mathcal{F}_{\mathbb{L} \rightarrow \Sigma_\perp^\#}$. Then*

$$\forall \ell \in \mathbb{L}, \forall x \in \mathbb{X} : \forall v \text{ free variable of } p^\#[\ell][x] : \text{loc}(v) \text{ dominates } \ell$$

PROOF. Equation (2) remains true, and we can iterate it again. □

THEOREM D.5 (THEOREM 5.3 IN THE POST-FIXPOINT CASE). *Let $p^\#$ be a post-fixpoint of $\mathcal{F}_{\mathbb{L} \rightarrow \Sigma_\perp^\#}$ for a program whose graph is \mathcal{G} . Then:*

$$\forall \ell \in \mathbb{L} : \forall x \in \mathbb{X} : \ell \in \text{scope}_{\mathcal{G}}(p^\#[\ell][x])$$

PROOF. Direct consequence of Theorem D.4 and definition of scope. □

LEMMA D.6 (LEMMA C.1 IN THE POST-FIXPOINT CASE). *Let $p^\#$ be a post-fixpoint of $\mathcal{F}_{\mathbb{L} \rightarrow \Sigma_\perp^\#}$ and $\hat{g} = \text{graph}(p^\#)$. Let $\ell' \in \mathbb{L}$ and $\ell \in \text{preds}(\ell')$. Let $\sigma^\# = p^\#[\ell]$ and $\sigma'^\# = p^\#[\ell']$. Let $\Gamma \mapsto \sigma \in \gamma_{\Sigma^\#}(\sigma^\#)$, i.e. $\sigma = \gamma_{\Sigma^\#}(\sigma^\#)[\Gamma]$.*

Then:

$$\{\sigma' : (\ell, \sigma) \rightsquigarrow (\ell', \sigma')\} = \{\gamma_{\Sigma^\#}(\sigma'^\#)[\Gamma'] : (\ell, \Gamma) \rightsquigarrow_{\hat{g}} (\ell', \Gamma')\} \quad (12)$$

$$\{(\ell', \sigma') : (\ell, \sigma) \rightsquigarrow (\ell', \sigma')\} = \{\mathcal{H}_{p^\#}(\ell', \Gamma') : (\ell, \Gamma) \rightsquigarrow_{\hat{g}} (\ell', \Gamma')\} \quad (13)$$

PROOF. The proof is similar to the one of Lemma C.1. Because we are a post-fixpoint instead of a fixpoint, there may be more " ϕ " variables created than necessary, but they are still compensated by a corresponding binding in the graph.

Moreover, Theorem 5.3 still holds when $p^\#$ is a post-fixpoint instead of a fixpoint (see Theorem D.5). Thus, the Lemma is still true when $p^\#$ is a post-fixpoint. \square

THEOREM D.7 (SOUNDNESS AND COMPLETENESS OF THE SSA ABSTRACTION COMPUTATION (THEOREM C.10) WHEN $p^\#$ IS A POST-FIXPOINT). *Let $p^\#$ be a post-fixpoint of $\mathcal{F}_{\mathbb{L} \rightarrow \Sigma_\perp^\#}$ and $\hat{g} = \text{graph}(p^\#)$.*

Then:

$$\gamma_{\text{SSA}^\#}(p^\#, \hat{g}) = \gamma_{\text{S}^\#}$$

PROOF. It suffices to redo all the proofs of Theorem C.10 starting from Lemma D.6. \square

D.2 Fully-propagated subgraph

In this section, we provide and prove equivalent two characterizations of a fully-propagated subgraph G :

- In one, G is the set of edges (ℓ, ℓ') which are stable when $\mathcal{F}_{\mathbb{L} \rightarrow \Sigma_\perp^\#}$ is applied, i.e. the edges for which $p^\#[\ell']$ represents the successors of the states represented by $p^\#[\ell]$;
- In the other, G is the part of the original program graph \mathcal{G} for which $p^\#$ is a post-fixpoint.

The first characterization provides a computable definition of a fully-propagated subgraph, while the second will allow reusing the Theorems of the previous section.

First we introduce some notations. A *subgraph* $G \in \mathcal{P}(\mathbb{L} \times \mathbb{L})$ is a set of edges. We denote by:

- $\mathbb{L}_{|G} \triangleq \{\ell \mid \exists \ell' : (\ell, \ell') \in G \vee (\ell', \ell) \in G\}$ the set of locations that appear in G ;
- $p^\#_{|G} \triangleq \{\ell \mapsto p^\#[\ell] \mid \ell \in \mathbb{L}_{|G}\}$ the restriction of $p^\# \in \mathbb{L} \rightarrow \Sigma_\perp^\#$ to the locations of G ;
- $\mathcal{G}_{|G} \triangleq \{(\ell, \ell') \mapsto e \mid (\ell, \ell') \mapsto e \in \mathcal{G} \wedge (\ell, \ell') \in G\}$ the restriction of the original program graph \mathcal{G} to the edges of the subgraph G ;
- $\mathcal{F}_{\mathbb{L} \rightarrow \Sigma_\perp^\#}^{|G}(p^\#) \triangleq \left[\ell' \in \mathbb{L} \mapsto \begin{cases} [x \in \mathbb{X} \mapsto x_{\ell_0}] & \text{if } \ell' = \ell_0 \\ \sqcup_{\ell' \in \text{preds}_G(\ell')} \{I^\#[[\text{instr}[\ell]](p^\#[\ell]) \mid \ell \in \text{preds}_G(\ell')\} & \text{if } \ell' \neq \ell_0 \end{cases} \right]$ where $\text{preds}_G(\ell') \triangleq \{\ell \in \mathbb{L} \mid (\ell, \ell') \in G \wedge \mathcal{G}[\ell, \ell'] \neq 0 \wedge p^\#[\ell] \neq \perp\}$ the restriction of the transfer function $\mathcal{F}_{\mathbb{L} \rightarrow \Sigma_\perp^\#}$ to $\mathcal{G}_{|G}$.

THEOREM D.8. *Let $p^\# \in \mathbb{L} \rightarrow \Sigma_\perp^\#$ and $G \subseteq \text{dom } \mathcal{G}$. The following conditions are equivalent:*

- *The edges of G are propagated in $p^\#$:*

$$\forall (\ell, \ell') \in G : \mathcal{G}[\ell, \ell'] \neq 0 \wedge p^\#[\ell] \neq \perp \Rightarrow I^\#[[\text{instr}[\ell]](p^\#[\ell])] \sqsubseteq^{\ell'} p^\#[\ell']. \quad (14)$$

- *$p^\#_{|G}$ is a post-fixpoint of $\mathcal{F}_{\mathbb{L} \rightarrow \Sigma_\perp^\#}^{|G}$:*

$$p^\#_{|G} \sqsubseteq \mathcal{F}_{\mathbb{L} \rightarrow \Sigma_\perp^\#}^{|G}(p^\#_{|G}) \quad (15)$$

When either holds, we say that G is a fully-propagated subgraph of $p^\#$.

PROOF.

- (15) \Rightarrow (14): If $(\ell, \ell') \in G$ then, by definition of $\mathcal{F}_{\mathbb{L} \rightarrow \Sigma_\perp^\#}^{|G}$, $\exists \sigma^\# : p^\#[\ell'] = I^\#[[\text{instr}[\ell]](p^\#[\ell]) \sqcup \sigma^\#$. This means that $I^\#[[\text{instr}[\ell]](p^\#[\ell])] \sqsubseteq^{\ell'} p^\#[\ell']$ per Definition 6.1.
- (14) \Rightarrow (15): A standard property of semi-lattices is $\forall a, b, c : a \sqsubseteq b \wedge a \sqsubseteq c \Leftrightarrow a \sqsubseteq (b \sqcup c)$. Using the definition of $\mathcal{F}_{\mathbb{L} \rightarrow \Sigma_\perp^\#}^{|G}$, this allows to conclude.

□

The following theorem provides a computable definition of a suitable fully-propagated subgraph G :

COROLLARY D.9. Let $p^\# \in \mathbb{L} \rightarrow \Sigma_\perp^\#$. The set

$$\max G(p^\#) \triangleq \{(\ell, \ell') \in \text{dom } \mathcal{G} \mid \mathcal{G}[\ell, \ell'] = 0 \vee p^\#[\ell] = \perp \vee \mathcal{I}^\#[\text{instr}[\ell]](p^\#[\ell]) \sqsubseteq^{\ell'} p^\#[\ell']\}$$

is the maximal fully-propagated subgraph of $p^\#$.

Note that the theorems in this section are not restricted to $p^\# \in \mathbb{L} \rightarrow \Sigma_\perp^\#$ and works for any flow-sensitive abstraction; in particular it can be readily extended in the case where $p^\#$ is combined with other abstract domains (e.g., to have a more precise discovery of dead locations and branches).

D.3 An incrementally complete transfer function

We can now complete the proof:

THEOREM D.10 (INCREMENTAL SSA GRAPHS ARE COMPLETE).

Let $p^\# \in \mathbb{L} \rightarrow \Sigma_\perp^\#$, $G = \max G(p^\#)$, and $\hat{g} = \text{graph}(\mathcal{F}_{\mathbb{L} \rightarrow \Sigma_\perp^\#}(p^\#))$. Then: $\gamma_{\text{SSA}^\#}(p^\#, \hat{g}|_G) \subseteq \gamma_{|\mathbb{S}^*}$.

PROOF. We define $\mathbb{S}_{|G} = \{(\ell, \sigma) \in \mathbb{S} \mid \ell \in \mathbb{L}_{|G}\}$.

$$\begin{aligned} \gamma_{\text{SSA}^\#}(p^\#, \hat{g}|_G) &= \gamma_{\text{SSA}^\#}(p^\#_{|G}, \hat{g}|_G) && \text{(by definition of } \gamma_{\text{SSA}^\#}\text{)} \\ &= \gamma_{\text{SSA}^\#}(p^\#_{|G}, \text{graph}(p^\#_{|G})) && \text{(by definition of the graph operator)} \\ &= \gamma_{|\mathbb{S}_{|G} \cap \mathbb{S}^*} && \text{(by Theorem D.7 and Theorem D.8)} \\ &\subseteq \gamma_{|\mathbb{S}^*} && \text{(by definition of restriction)} \end{aligned}$$

□

Note that if the intermediate versions of the SSA graph produced by $\mathcal{F}'_{\text{SSA}^\#}$ are now complete, $\mathcal{F}'_{\text{SSA}^\#}$ is not monotonic, because $\max G$ is not monotone: if $\mathcal{F}_{\mathbb{L} \rightarrow \Sigma_\perp^\#}(p^\#)$ always represents more states than $p^\#$, there may be fewer fully-propagated locations in $(\mathcal{F}_{\mathbb{L} \rightarrow \Sigma_\perp^\#}(p^\#))$ than in $p^\#$.

E USE CASE : SINGLE-PASS OPTIMIZED TRANSLATIONS TO SSA

This section describes our experiments to use SSA translation based on abstract interpretation to compile (a subset of) C to the LLVM IR (which is in SSA form).

Implementation. Our goal is to demonstrate the feasibility of using dataflow analysis as a basis for SSA translation. We have implemented the algorithms described in the paper as a small plugin for the FRAMA-C platform, and used the LLVM intermediate representation as the target of our translation. In particular, we have implemented the symbolic expression analysis of Section 4, followed by one call to the graph operator, modified to directly produce LLVM code instead of SSA graphs. This second, linear, pass consists on traversing all the subterms in the global value graph, determining for each the location where to place the instruction, and then inserting the corresponding instruction in the LLVM IR.

We have experimented with several possibilities for finding where to insert instructions. The first is to use our scope function. This corresponds to the "schedule early" strategy of Click [1995]. An alternative is change our symbolic expressions so that scope becomes explicit; i.e. the language of symbolic expression becomes $e \in \mathbb{E} \triangleq \mathbb{L} \times (z \mid v \mid e \diamond e)$. It is easy to change the analysis to register the current location in the newly created expressions. The benefit of this technique is that the SSA

	Clang	With Constant Propagation			Without Constant Propagation		
		Iterative (/Clang)	Recursive (/Clang)	$\hat{G} \rightarrow \text{LLVM}$	Iterative (/Clang)	Recursive (/Clang)	$\hat{G} \rightarrow \text{LLVM}$
Min	0.06s	0.207s ($\times 1.70$)	0.205s ($\times 1.66$)	0s	0.205s ($\times 2.36$)	0.204s ($\times 2.34$)	0s
Median	0.29s	0.601s ($\times 2.19$)	0.612s ($\times 2.19$)	0.027s	0.814s ($\times 3.30$)	0.791s ($\times 3.27$)	0.284s
Average	0.359s	0.754s ($\times 2.48$)	0.759s ($\times 2.48$)	0.365s	1.230s ($\times 3.32$)	1.220s ($\times 3.30$)	1.084s
Max	1.2s	2.270s ($\times 3.70$)	2.343s ($\times 3.73$)	0.127s	4.572s ($\times 5.57$)	5.756s ($\times 5.33$)	7.947s

Table 1. Time spent in SSA translation in the generated programs (and compared to a standard compiler run).

translation is closer to original program and to what a usual SSA translation would perform; the drawback is that the global value numbering detects fewer equalities between symbolic variables. The time below is given for this alternative version.

A necessary change is to insert terminator instructions (branches and conditional branches) instead of using guards and non-determinism. Formally, we define terminators as $term \in Terminator \triangleq Exit \mid \text{Jump}(\ell) \mid \text{JumpIf}(e, \ell, \ell)$, we replace the graph of the original program by a mapping $\mathbb{L} \rightarrow Terminator$ (using a program expression for the condition), and we replace the guards on the SSA graphs by a mapping $\mathbb{L} \rightarrow Terminator$ (using symbolic expression for the condition). The semantics is the standard one.

The implementation of this plugin, that does SSA translation, Global value numbering, global code motion, and sparse conditional constant propagation, required only 970 lines of OCaml code (measured by Sloccount), excluding a 316 lines implementation of Okasaki's Patricia tries [Okasaki and Gill 1998], and code implementing the iterative and recursive iteration strategy on Bourdoncle's [Bourdoncle 1993] WTO (183 lines). The parser and WTO calculation is provided by FRAMA-C, and the LLVM translation by the standard LLVM OCaml bindings.

Evaluation. We have used this implementation to evaluate both the soundness and performance of our algorithm. In languages like C, the variables are addressable, in which case they cannot be directly translated to SSA (Clang, for instance, does not translate C variables to SSA variables and delegates this process to later passes). To evaluate our algorithm on large programs containing only non-addressable variables, we have used the CSMITH [Yang et al. 2011] C generator, using options to generate only non-addressable integer variables (but complex unstructured control flow), to generate 100 large random programs (in particular, we increased CSMITH option `-max-block-size` to 10 to generate larger files). The median size of the files is 144kb, the largest is weighting 692kb. These are very large C file, and the FRAMA-C parser (whose performance are usually acceptable) took more than 5 minutes just to parse some of these files.

We used the FRAMA-C parser on this program, our dataflow analysis to generate an SSA graph followed by a linear pass to translate it to the LLVM Intermediate Representation; finally we used LLVM (version 13) to compile the result to native x86 code, that we executed. We then compared the result to a native LLVM compilation. All tests were executed on a standard laptop (Intel i7-8850H CPU @ 2.60GHz). The code for the experiment and its results can be retrieved at <https://doi.org/10.5281/zenodo.7152678>.

Soundness. We obtained the following results: all the 100 files provided the same result as the execution of the LLVM version. In another run we generated 100 other files without the safe-math option of CSMITH (that add guards before undefined behaviors, like overflowing shifts or division by zero), in which case we found a difference for 18 files, all of them explainable by an actual undefined behavior in the code (this was checked using the FRAMA-C/EVA [Bühler et al. 2017] abstract interpreter).

Performance. We measured the time needed for the dataflow analysis pass (Table 1) in 4 configurations : by choosing a fixpoint iteration strategy [Bourdoncle 1993] (Recursive or Iterative), or by

1961 enabling or disabling the combination of the SSA domain with constant propagation and dead code
1962 elimination. We provide this time in seconds, but also as a ratio compared to the performance of a
1963 standard compiler (using clang to compile the file). For instance, at worst the iterative strategy with
1964 constant propagation took 2.270s, and at worst it took 3.70 times the time spent by clang to compile
1965 the file. We can observe that the choice between different iteration strategies does not significantly
1966 impact the performance of the analysis, contrary to the use of constant propagation and dead
1967 code elimination. Furthermore, in a previous version of the program, we used the standard Map of
1968 OCaml instead of Okasaki maps to implement our abstract stores (as done in Astrée [Blanchet et al.
1969 2002]), in which case the performance was noticeably slower (e.g. the analysis could take up to one
1970 minute).

1971 We also measured the time for the linear compilation pass translating our SSA graphs to LLVM IR
1972 (with and without constant propagation, as the choice of an iteration strategy does not impact the
1973 result of the dataflow analysis). Again, the use of constant propagation and dead code elimination
1974 allows spending less time in this phase. The time spent in this phase is generally low except in
1975 some cases; a large part of this is due to a naive translation generating one LLVM basic block per C
1976 instruction. The majority of the work in this phase is spent in LLVM, and it is probable that the
1977 exported OCaml bindings do not provide the most efficient way of building the program in the
1978 intermediate representation.

1979 Finally, we also measured the maximum number of iterations required to attain the fixpoint
1980 when analyzing a strongly connected component, which was 6 in the case of an iterative strategy
1981 and 3 for the iterative strategy. This contrasts with the theoretical maximum, which corresponds to
1982 the number of program variables (usually several thousands in our tests).

1983 *Conclusions.* Doing SSA translation using a dataflow analysis is a sound and simple way to perform
1984 SSA translation. The soundness is not a surprise since we provided the proof for all of our algorithms,
1985 but the experimental evaluation confirms that it works in practice.

1986 Furthermore, *the time spent doing so is in the order of magnitude compatible for incorporation in a*
1987 *compiler:* if we add incorporated our SSA translation pass inside a compiler, the maximum observed
1988 slowdown would have been $\times 6.57$ (i.e. $5.57 + 1$), which is slower but acceptable. This is a maximum,
1989 as such an incorporation would lead to removing existing passes in clang; furthermore, additional
1990 engineering (such as switching from OCaml to C) could further improve the performance. *One of*
1991 *the reason for this efficiency is that fixpoint iteration with the symbolic expression domain is usually*
1992 *obtained using few iterations,* even in large programs. The other part is explained by the use of
1993 functional maps for low-complexity operations on abstract stores. *One crucial engineering decision*
1994 *in our analysis performance was the use of Okasaki maps [Okasaki and Gill 1998] to implement the*
1995 *abstract stores,* instead of the more common binary search trees [Blanchet et al. 2002], as otherwise
1996 the time spent in the \sqcup^f operation dominates the time spent in the analysis. Okasaki maps allow
1997 more efficient handling of the parts of the abstract store that have not changed between when the
1998 different states that are joined with \sqcup^f . Finally, *combining SSA translation with other analyses such*
1999 *as constant propagation and dead code elimination can significantly improve the time spent in the*
2000 *analysis:* in this case, incorporating our pass in LLVM would make it only 4.73 time slower (instead
2001 of 6.57 times), at worst.

2002 *Whether, with careful engineering, implementing compilers as a single-pass combination of modular*
2003 *abstract domains, can outperform the existing designs as a sequence of passes, is an open question,* but
2004 our initial results indicate that the single-pass design could be competitive in terms of speed of
2005 compilation.
2006
2007
2008
2009